Università
degli Studi
di Ferrara

DEPARTMENT OF ENGINEERING

DOCTORAL DEGREE IN ENGINEERING SCIENCE

Cycle XXXVIII

Coordinators: Prof. Stefano Trillo and Prof. Pier Ruggero Spina

Scalable, probabilistic, and explainable Machine Learning

Scientific-Disciplinary Sector ING-INF/05

Candidate

Dott. Elisabetta Gentili

Supervisor

Prof. Evelina Lamma

Co-Supervisor

Prof. Fabrizio Riguzzi

Years 2022 - 2025

Abstract

In the field of Artificial Intelligence (AI), Machine Learning (ML) comprises automated data analysis techniques aimed at learning models directly from data. These include classification and prediction models, as well as clustering approaches for grouping data. In recent years, the rapid growth of the availability of massive volumes of digital information has expanded the range of ML applications, while also raising new challenges. For example, datasets are often incomplete, inconsistent, or uncertain. To address these issues, ML algorithms must remain scalable as data volumes increase, and must also be able to handle imperfect or missing information and unbalanced class distributions. Probabilistic ML methods provide a natural way to model uncertainty in such settings, while explainable models ensure that the outcomes of ML systems can be understood and trusted even by non-experts. Inductive Logic Programming (ILP), a subfield of ML, produces rule-based models that are intrinsically interpretable, and thus represents a promising approach for explainable and trustworthy AI. Logic-based approaches offer suitable tools in this context, as they support structured knowledge representation, probabilistic modeling, and interpretable inference. In this context, this thesis explores the application of both logic-based and traditional ML methods to diverse domains.

The first contribution concerns the development of probabilistic logic systems. One is LIFTCOVER+, an enhanced ILP algorithm that performs both structure and parameter learning. Structure learning identifies logical rules that best explain the observed examples given the background knowledge, while parameter learning assigns probabilistic weights to the rules. In addition, a second methodological contribution addresses parameter learning in Probabilistic Answer Set Programming (PASP), broadening the range of probabilistic reasoning tools available in ILP.

The second contribution lies in demonstrating the applicability and effectiveness of ML and ILP methods across real-world scenarios, including knowledge graph comple-

tion, financial services, and medical risk prediction.

The third contribution explores the combination of logic and large language models (LLMs) to improve the interpretability of logical theories, especially when they include invented predicates lacking meaningful names.

Sommario

Nel campo dell'Intelligenza Artificiale (IA), il Machine Learning (ML) comprende tecniche di analisi automatizzata dei dati finalizzate all'apprendimento di modelli direttamente dai dati. In particolare include modelli di classificazione e previsione, nonché approcci di clustering per raggruppare i dati. Negli ultimi anni, la rapida crescita della disponibilità di enormi volumi di informazioni digitali ha ampliato il campo delle applicazioni di ML, sollevando però anche nuove sfide. Ad esempio, i dataset sono spesso incompleti, inconsistenti o incerti. Per affrontare questi problemi, gli algoritmi di ML devono rimanere scalabili con l'aumento del volume dei dati e devono anche essere in grado di gestire informazioni sbagliate o mancanti e distribuzioni di classi sbilanciate. I metodi probabilistici di ML offrono un modo naturale per modellare l'incertezza in questi contesti, mentre i modelli spiegabili garantiscono che il comportamento e i risultati dei sistemi di ML possano essere compresi e affidati anche da utenti non esperti. L'Inductive Logic Programming (ILP), parte del ML, produce modelli basati su regole logiche che sono intrinsecamente interpretabili e rappresenta quindi un approccio spiegabile e affidabile. Gli approcci basati sulla logica offrono strumenti adeguati in questo contesto, poiché supportano la rappresentazione strutturata della conoscenza, la modellizzazione probabilistica e l'inferenza interpretabile. In quest'ambito, questa tesi esplora l'applicazione di metodi basati sulla logica e metodi tradizionali di ML in diverse applicazioni.

Il primo contributo è LIFTCOVER+, un algoritmo di ILP per l'apprendimento della struttura e dei parametri di programmi logici probabilistici. L'apprendimento della struttura identifica le regole logiche che meglio spiegano gli esempi data una conoscenza di background, mentre l'apprendimento dei parametri assegna pesi probabilistici alle regole. Inoltre, un secondo contributo metodologico riguarda l'apprendimento dei parametri nel Probabilistic Answer Set Programming (PASP), ampliando la gamma di strumenti di ragionamento probabilistico disponibili nell'ILP.

Il secondo contributo riguarda l'analisi dell'applicabilità e dell'efficacia di metodi di ML e ILP in diversi scenari reali, tra cui la knowledge graph completion, il supporto decisionale in ambito finanziario e la previsione di fattori di rischio in ambito medico.

Il terzo contributo esplora la combinazione di logica e modelli di linguaggio di grandi dimensioni (LLM) per migliorare l'interpretabilità delle teorie logiche, specialmente quando queste includono predicati inventati privi di nomi significativi.

Acknowledgements

First of all, I would like to thank my supervisors, Prof. Evelina Lamma and Prof. Fabrizio Riguzzi, for their continuous guidance and support during my PhD and the numerous opportunities they gave me.

I am very grateful to Prof. Katsumi Inoue for his guidance and support during the six months I spent at the National Institute of Informatics, in Tokyo, Japan.

I also thank Dr. Maria Ferrara and Ilaria Domenicano, PhD, from the Department of Neuroscience and Rehabilitation of the University of Ferrara, for giving me the opportunity to apply my knowledge within the psychiatric field.

I thank the AI@UNIFE research group and my lab colleagues, for the exchanges of ideas and nice chats.

And last but not least, I thank my family for the continuous support during these years.

Declarations

This thesis was produced while attending the PhD programme in Engineering Science at the University of Ferrara, Cycle XXXVIII, with the support of a scholarship financed by the Ministerial Decree no. 351 of 9th April 2022, based on the NRRP - funded by the European Union - NextGenerationEU - Mission 4 “Education and Research”, Component 1 “Enhancement of the offer of educational services: from nurseries to universities” - Investment 4.1 “Extension of the number of research doctorates and innovative doctorates for public administration and cultural heritage”.

Ishi no ue ni
mo sannen.

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence

AMR Abstract Meaning Representation

ASP Answer Set Programming

AUC Area Under the Curve

BDD Binary Decision Diagram

CoT Chain-of-Thought

CV Cross Validation

FM Foundation Model

FN False Negative

FOL First-Order Logic

FP False Positive

FPR False Positive Rate

HPLP Hierarchical Probabilistic Logic Program

ICD-9 International Classification of Diseases - 9th revision

ILP Inductive Logic Programming

KG Knowledge Graph

KGC Knowledge Graph Completion

LAIs long-acting injection

LFF Learning From Failures

LFIT Learning From Interpretation Transitions

LLM Large Language Model

LOO Leave-One-Out

LP Logic Programming

LPADs Logic Programs with Annotated Disjunctions

LPO Leave-P-Out

ML Machine Learning

MSE Mean Squared Error

NNF Negation Normal Form

PI Predicate Invention

PILP Probabilistic Inductive Logic Programming

PLP Probabilistic Logic Programming

PPV Positive Predictive Value

PRC Precision-Recall curve

RL Reinforcement learning

ROC Receiver Operating Characteristic

SVM Support Vector Machine

TII Technology Innovation Institute

TN True Negative

TNR True Negative Rate

TP True Positive

TPR True Positive Rate

VIF Variance Inflation Factor

WFF Well-Formed Formula

Part I

Introduction

Chapter 1

Motivation

Even though Artificial Intelligence (AI) has gained widespread public attention only in recent years, its origins dates back to the 1950s. Since then, AI has undergone alternating phases of optimism and scepticism. In the past decade, however, the field has experienced a substantial and continuous growth, driven primarily by significant advances in computational power and by the availability of large amounts of data. As a result, nowadays AI-based systems are continuously employed across a wide range of application domains, such as healthcare, finance, robotics, gaming, manufacturing, and transportation, demonstrating the broad applicability and impact of AI technologies. In general, AI involves techniques and methodologies aimed at designing systems capable of carrying out tasks typically associated with human intelligence, such as learning from experience, solving complex problems, or processing natural language.

AI comprises several subfields. One of the most prominent is Machine Learning (ML), which refers to methods enabling computational systems to automatically learn from data and improve their performance with experience, under different learning paradigms. Despite ML's widespread adoption and success in many domains, a substantial part of human reasoning relies on relational structures and explicit knowledge representation, aspects that purely data-driven methods do not always capture effectively. This limitation motivates the exploration of complementary paradigms that combine learning with symbolic reasoning.

A longstanding research area that addresses this need is logic programming. Logical formalisms provide a powerful and expressive framework for modeling domains characterized by structured or relational information. In particular, First-Order Logic

(FOL) offers a well-established foundation for representing entities and the relationships between them. Logic Programming (LP) builds upon this foundation to define knowledge in terms of facts and rules, enabling transparent and interpretable reasoning. Probabilistic Logic Programming (PLP) extends LP by incorporating uncertainty directly into the logical representation, allowing systems to reason with both symbolic structures and probabilistic information.

A research direction that formalizes this integration is Inductive Logic Programming (ILP). Founded by Muggleton in the 1990s, ILP is a field that lies at the intersection of ML and knowledge representation. One of its distinctive advantages is its ability to construct hypotheses from a limited number of examples, making it particularly suitable in scenarios where data are scarce. Moreover, ILP systems leverage the expressive power of logic programs to handle relational and structured data, a setting in which traditional ML methods often struggle. The logic-based representation also supports the seamless incorporation of prior knowledge, enabling models to reason in ways that are both guided and constrained by expert understanding. Because hypotheses are expressed in human-readable logical form, ILP is naturally explainable, providing clear information on the behavior of the models and their outputs.

These motivations underpin the research presented in this thesis.

Chapter 2

Thesis Aims

The primary aim of this thesis is to investigate methods for both structure and parameter learning within logic-based frameworks, and to explore their application for solving complex real-world problems. Specifically, the thesis addresses three main objectives:

Probabilistic ILP systems for structure and parameter learning. The first goal is to improve existing ILP methods and propose new ones for structure and parameter learning. This is achieved through the development of LIFTCOVER+, an algorithm designed to identify rules that best explain observed examples while assigning meaningful probabilistic weights, thus providing both predictive accuracy and interpretable logical models. This goal also includes broadening the range of probabilistic reasoning tools available within ILP, with a particular focus on parameter learning in Probabilistic Answer Set Programming (PASP), exploring different approaches. Two of these approaches rely on extracting symbolic equations that represent the probabilities of interpretations: one solves the task using an off-the-shelf constrained optimization solver, while the other employs an implementation of the Expectation-Maximization algorithm. With a third one, we introduce the gradient semiring for PASP and implement it on top of a state-of-the-art solver.

Explore the application of ML and ILP systems in different domains. The approaches we proposed were tested on benchmark datasets, showing performances comparable and in some cases even superior to those of state-of-the-art systems, motivating their application across multiple domains.

LIFTCOVER+ was used to solve the task of Knowledge Graph Completion (KGC), which involves inferring missing information in an incomplete Knowledge Graph that represents knowledge on a specific domain. We investigated three different approaches for performing parameter learning of rules that predict this missing information using LIFTCOVER+, with the goal of evaluating whether parameter learning can be a competitive approach for solving the KGC task.

We also used LIFTCOVER+ as a decision-support system in a financial setting. The expressive and interpretable nature of LIFTCOVER+'s output proved to be particularly valuable in this context, as it facilitated the integration of the model into a setting with users who may not have a background in formal logic. This characteristic not only enhanced the model's accessibility but also enabled the users to make informed decisions based on understandable results.

Using traditional ML methods, we also conducted an analysis in the medical field, based on data collected from psychiatric patients in the Ferrara province, to identify risk factors of medication non-adherence. The ML pipeline begins with feature preprocessing, followed by the training of several predictive models. We evaluate their performance with cross-validation to ensure robust results. Finally, we identify the specific risk factors by analyzing the coefficients of a logistic regression model, providing interpretable insights into the factors that contribute to medication non-adherence.

Improving interpretability of logical theories with large language models. The third goal is to investigate the use of Large Language Models (LLMs) to enhance the readability and semantic meaningfulness of logical theories, especially when they involve invented predicates that lack intuitive names. This integration seeks to combine the expressiveness of symbolic logic with the natural language understanding capabilities of LLMs.

Together, these objectives aim to advance the methodological foundations of ML and probabilistic ILP, while demonstrating their applicability to real-world problems.

Chapter 3

Thesis Structure and Contributions

The content of this thesis is organized into the following parts and chapters.

Part I Introduction

This part outlines the motivation, aims, and contributions of the thesis. It also includes an overview of its structure, presented in the current chapter.

Part II Foundations

This part is divided into three chapters, each presenting the fundamental concepts and theoretical background that form the basis of the contributions and applications presented in this thesis.

Specifically, Chapter 4 provides an overview of Machine Learning. It briefly introduces the main learning paradigms, namely supervised, unsupervised, and reinforcement learning, together with their specific tasks and learning objectives. The chapter also reviews the most commonly used ML models and the evaluation metrics typically applied in practice. In addition, it outlines the standard ML pipeline, covering data preprocessing, model selection, training, and evaluation. Finally, it provides a brief introduction to large language models and discusses some commonly used prompting strategies.

Chapter 5 introduces the fundamental concepts of propositional and first-order logic, in terms of both syntax and semantics. Then it introduces the principles of logic programming. Finally, it presents probabilistic logic programming, with particular

emphasis on the distribution semantics, explaining how probabilistic reasoning is integrated into logic programs and how uncertainty can be modeled within this framework.

Chapter 6 introduces Inductive Logic Programming, describing its two main learning settings and outlining how ILP systems explore the hypothesis space to induce logical rules that generalize the observed data. The chapter then presents Probabilistic ILP (PILP), which extends ILP by incorporating probabilistic reasoning to handle uncertainty. It describes the two main tasks in PILP: parameter learning, which estimates the probabilities associated with rules, and structure learning, which identifies the structure of the rules.

Part III - Probabilistic Learning Systems

Building on the theoretical foundations introduced in Part II, this part focuses on probabilistic learning systems. In particular, Chapter 7 presents LIFTCOVER+, an ILP system designed to perform both parameter and structure learning for liftable probabilistic logic programs. The system's principles and capabilities are discussed, and the results of its evaluation on benchmark datasets are presented.

The content of this chapter is based on:

- E. Gentili, A. Bizzarri, D. Azzolini, R. Zese, and F. Riguzzi, "Regularization in probabilistic inductive logic programming," in *International Conference on Inductive Logic Programming*, pp. 16–29, Springer, 2023. doi:10.1007/978-3-031-49299-0_2
  Best Student Paper Award at the 32nd International Conference on Inductive Logic Programming ILP2023 @ IJCLR.

Then, Chapter 8 presents different approaches to address the parameter learning task within the Probabilistic Answer Set Programming framework.

The content of this chapter is based on:

- D. Azzolini, E. Gentili, and F. Riguzzi, "Symbolic parameter learning in probabilistic answer set programming," *Theory and Practice of Logic Programming*, vol. 24, no. 4, pp. 698–715, 2024. doi:10.1017/S1471068424000334

- E. Gentili, A. Bizzarri, D. Azzolini, and F. Riguzzi, "The gradient semiring for probabilistic answer set programming and its application to parameter learning,"

in *Proceedings of the 5th International Joint Conference on Learning & Reasoning*, 2025. To appear.

Part IV - Applications

This part presents several applications of LIFTCOVER+ and traditional Machine Learning models in different contexts.

In Chapter 9, we employ LIFTCOVER+ for solving the task of Knowledge Graph Completion. The content of this chapter is based on:

- E. Gentili, "Knowledge graph completion with probabilistic logic programming," *Proceedings of the AIxIA Doctoral Consortium*, 2023. https://ceur-ws.org/Vol-3670/paper96.pdf

- D. Azzolini, M. Bonato, E. Gentili, F. Riguzzi, "Logic programming for knowledge graph completion," *39th Italian Conference on Computational Logic (CILC 2024)*, in *CEUR Workshop Proceedings*, vol. 3733, pp. 1–14, CEUR-WS, 2024. https://ceur-ws.org/Vol-3733/paper4.pdf

- D. Azzolini, E. Gentili, F. Riguzzi, "Link prediction in knowledge graphs with probabilistic logic programming: Work in progress," *Workshop on Probabilistic Logic Programming (PLP 2023)*, in *CEUR Workshop Proceedings*, vol. 3437, pp. 1–4, CEUR-WS, 2023. https://ceur-ws.org/Vol-3437/short5PLP.pdf

Chapter 10 illustrates the application of LIFTCOVER+ to an industrial context, where it was employed as a predictive model to support a financial company in optimizing its interactions with clients.

Chapter 11 illustrates a pipeline using traditional machine learning predictive models to identify risk factors for medication non-adherence.

Chapter 12 presents two datasets from the Neuro-Symbolic (NeSy) field, with which we contributed to the development of a benchmark for evaluating the capabilities of NeSy systems. The content of this chapter is based on:

- R. Manhaeve, F. Giannini, M. Ali, D. Azzolini, A. Bizzarri, A. Borghesi, S. Bortolotti, L. De Raedt, D. Dhami, M. Diligenti, S. Dumančić, B. Faltings, E. Gentili, A. Gerevini, M. Gori, T. Guns, M. Homola, K. Kersting, J. Lehmann, M. Lombardi, L. Lorello, E. Marconato, S. Melacci, A. Passerini, D. Paul, F. Riguzzi,

S. Teso, N. Yorke-Smith, and M. Lippi, "Benchmarking in neuro-symbolic ai," in *Learning and Reasoning: 4th International Joint Conference on Learning and Reasoning, IJCLR 2024, and 33rd International Conference on Inductive Logic Programming, ILP 2024, Nanjing, China, September 20–22, 2024, Proceedings*, W.-Z. Dai, Ed. Cham: Springer Nature Switzerland, 2026, pp. 238–249.

Lastly, in Chapter 13 we address the challenge of assigning names to predicates in logic rules that contain unnamed predicates using LLMs. Several ILP systems may produce rules with unnamed predicates, which often appear in the output. This hinders the readability, interpretability, and reusability of the produced logic theory. Leveraging recent advancements in the development of LLMs, we explore their potential to suggest meaningful names for these unnamed predicates, thereby improving the overall interpretability of the logic rules. The content of this chapter is based on:

- E. Gentili, T. Ribeiro, F. Riguzzi, and K. Inoue, "Predicate renaming via large language models," *arXiv preprint arXiv:2510.25517*, 2025.

Part V - Conclusions

The final part of this thesis provides a summary of the work carried out and draws conclusions.

Part II

Foundations

Chapter 4

Machine Learning Foundations

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) that refers to the development of systems that automatically learn from data and improve their performance through experience. More precisely, a system learns a task T from experience E with P as a performance measure, if its performance P on T improves with E [1].

4.1 Overview of paradigms

ML encompasses a variety of learning paradigms, each characterized by the nature of the data available to the learner and the type of feedback received during the learning process.

Supervised learning In supervised learning, the model is trained on a labeled dataset, commonly referred to as the training set. Each training example consists of an input instance and its corresponding label, which represents the desired output for that instance. The objective of the model is to learn a mapping from inputs to outputs that generalizes well to unseen data, thereby enabling accurate prediction of labels for new instances. Supervised learning tasks are generally divided into two major categories: classification and regression.

In a classification task, the goal is to assign each input to one of a finite set of discrete categories, called classes. A class represents a group of instances sharing common properties or characteristics. The learning algorithm is provided with examples labeled with their respective classes and produces a classifier that maps the feature space

(typically represented as attribute–value pairs) to class labels. Classification can be binary, when there are only two possible labels (e.g., "spam" or "not spam"), or multi-class, when the label set includes more than two categories (e.g., "red", "blue", or "green"). Once trained, the classifier predicts the class of new, unlabeled examples based on their attributes.

In contrast, regression addresses problems in which the output variable takes continuous values. The task consists of learning a real-valued function, commonly referred to as the regression function, which models the dependency between one or more independent (input) variables and a dependent (output) variable. Therefore, the task of regression is to learn a hidden relationship between x and y from observed and possibly noisy data points [2, 3].

Unsupervised learning In unsupervised learning, the training data are unlabeled, that is, no explicit target values or class labels are provided. The objective is to discover the underlying structure, regularities, or representations within the data itself. Unlike supervised learning, where the model is guided by known outcomes, unsupervised learning seeks to infer patterns directly from the input distribution, typically by identifying groups of similar instances or by learning compact representations.

The most common types of unsupervised learning tasks are clustering and dimensionality reduction.

In clustering, the goal is to partition a set of data points into subsets (or clusters) such that points within the same cluster are more similar to each other than to those in different clusters. One of the most used algorithm for clustering is k-means.

Dimensionality reduction, on the other hand, aims to project high-dimensional data into a lower-dimensional space while preserving as much of the intrinsic structure or variance as possible. Principal Component Analysis (PCA) is one of the most used techniques for dimensionality reduction.

Another example of unsupervised learning is represented by association rules, which can be extracted from datasets where each instance is represented as a set of items. An association rule has the general form $X \implies Y$, where X and Y are called itemsets. The rule states that the occurrence of X in an example (also called a transaction) implies, with a certain degree of probability, the occurrence of Y in the same transaction. Each association rule is typically characterized by two key measures: support, which quantifies how frequently the rule applies to the dataset, and confidence, which

estimates the conditional probability of Y given X.

Reinforcement learning Reinforcement learning (RL) differs fundamentally from both supervised and unsupervised paradigms, as learning occurs through interaction with an environment rather than from a fixed dataset. In this framework, an agent repeatedly observes the current state of the environment, selects an action, and receives a reward (or penalty) that reflects the immediate or long-term utility of that action. The agent's goal is to learn a mapping from states to actions that maximizes the reward over time [4].

These paradigms constitute the foundational learning frameworks of traditional ML. Each paradigm has a different concept of "experience" and "feedback", reflecting distinct approaches to the inductive inference process of ML systems.

4.2 Typical Machine Learning workflow

The typical ML workflow or pipeline consists of a sequence of interconnected steps going from data acquisition and preprocessing, feature selection, model selection, model training, and model evaluation. Each stage has a direct consequence on the following ones, making it essential to carry out every step carefully to ensure the reliability and effectiveness of the final model. This pipeline is inherently iterative, meaning that insights gained during evaluation often trigger revisions in earlier phases such as feature design or data preprocessing.

Data acquisition and preprocessing Often, data may come form different sources and possibly in heterogeneous formats, leading to inconsistent, missing, sparse, or wrong information, as well as outliers and noise. These are common problems that need to be addressed in the preprocessing phase, which usually includes actions such as cleaning, encoding, and normalization. Cleaned data is usually partitioned into subsets used in different stages of the learning process: training set for model training, validation set for hyperparameters tuning, and test set for the evaluation of the final model. Using different datasets throughout the learning process is essential to prevent overfitting, that is, when a model learns the training data too closely and fails to generalize to unseen data. The quality of this preprocessing phase is critical, as errors or biases introduced here can cascade and degrade downstream learning.

Feature engineering and representation Feature engineering involves transforming raw data into informative variables (features) that the learning algorithm can effectively exploit. This process may include the extraction or creation of new features, encoding categorical values, polynomial expansions, embeddings, or other transformations. A complementary step is feature selection, which automatically identifies a subset of relevant features. This step is crucial for several reasons: it discards non-informative variables, mitigates overfitting, reduces dimensionality and model complexity, thereby improving both efficiency and interpretability. [5]. The chosen representation implicitly defines (or constrains) the hypothesis space available to the learning algorithm and is thus deeply tied to the inductive bias of the model.

Model selection and training Once the data and features are prepared, the next step is to select one or more models to train. The goal is to build a model capable of performing well on new, unseen data. During training, the model's parameter space is explored to minimize a loss function on the training set. Hyperparameters are typically tuned using validation data. At this stage, inductive bias is manifested in choices of architecture, regularization, and model class.

Model evaluation The evaluation phase assesses how well the trained model generalizes to unseen data, and it must therefore be performed on data that was not used during training. However, when the original dataset is too small to be partitioned, further reducing the number of training examples can lead to underfitting, that is, the model fails to capture the underlying structure of the data. In such cases, an effective alternative is cross-validation, a technique that involves splitting the dataset into multiple subsets in different ways to make the most of the available data. Cross-validation not only maximizes the use of limited data but also helps reduce variability in performance estimates, leading to more robust model evaluation.

A widely used method is K-fold cross-validation, which divides the dataset into k equally sized folds. Each fold is used once as the validation set, while the remaining k-*1* folds serve as the training set. This process is repeated k times, ensuring that every example is used for validation exactly once.

Another used method is Leave-P-Out (LPO) cross-validation, where p examples are selected as the validation set, and the remaining data is used for training. This process is repeated for all possible combinations of p validation examples within the dataset.

A special case of this approach is Leave-One-Out (LOO) cross-validation, where *p=1*. Appropriate evaluation metrics are selected based on the task, a topic that will be addressed in a later section.

4.3 Evaluation

Model evaluation is a crucial step in assessing the quality of the learned model. The choice of metrics depends on the specific task; in this section, we present standard performance metrics such as accuracy, precision, recall, and F1-score, as well as methods for statistical validation. In addition to predictive performance, interpretability has emerged as an increasingly important property of ML models. This naturally leads to the introduction of symbolic and logic-based learning as an approach to inherently interpretable machine learning, which will be discussed in detail in the following chapters.

Metrics for Classification

At the end of the training process, each example in the dataset falls into one of four possible cases: it can be a true positive (TP) if a positive example is correctly classified as positive, a true negative (TN) if a negative example is correctly classified as negative, a false positive (FP) if a negative example is incorrectly classified as positive, or a false negative (FN) if a positive example is incorrectly classified as negative. These outcomes are typically summarized using a confusion matrix, which collects the counts of TP, TN, FP, and FN. A confusion matrix is thus a two-dimensional table that represents on one axis the true class labels, while the other corresponds to the predicted labels produced by the classifier.

These values are then used to evaluate the model's performance with different metrics:

- Accuracy measures the proportion of correct predictions:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

- Error rate measures the proportion of incorrect predictions:

$$Error\ Rate = \frac{FP + FN}{TP + FP + TN + FN} = 1 - Accuracy$$

- Precision or Positive Predictive Value (PPV) measures the proportion of positive examples found among all those classified as positive:

$$PPV = \frac{TP}{TP + FP}$$

- Recall or True Positive Rate (TPR) or Sensitivity measures the proportion of positive examples found among all the positive ones:

$$TPR = \frac{TP}{TP + FN}$$

- Specificity or True Negative Rate (TNR) measures the proportion of negative examples found among all the negative ones:

$$TNR = \frac{TN}{TN + FP}$$

- False Positive Rate (FPR) measures the proportion of misclassified negative examples among all the negative ones:

$$FPR = \frac{FP}{FP + TN}$$

- F1-score is the harmonic mean of precision and recall:

$$F1\text{-}score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

- Balanced Accuracy is the arithmetic mean of sensitivity and specificity, especially useful when dealing with imbalanced data:

$$Balanced\ Accuracy = \frac{Sensitivity + Specificity}{2}$$

In binary classification, models often produce a numeric score or probability to estimate how likely an example belongs to the positive class. To convert this score into a binary prediction (positive or negative), a decision threshold is applied. The threshold can be varied to influence the trade-off between different performance metrics and potentially improve the performance of the model. The Receiver Operating Characteristic (ROC) curve plots the TPR against the FPR across different threshold values. The Area Under the Curve (AUC) of the ROC curve provides an aggregate measure of the classifier's ability to distinguish between the two classes: the closer the AUC is to 1, the better the model is at discrimination. The Precision-Recall curve (PRC), on the other hand, shows the trade-off between precision and recall across varying thresholds. This curve is particularly useful when dealing with imbalanced datasets, where the positive class is rare. A high AUC of the PRC indicates that the model achieves both high precision (returning mostly correct positive predictions) and high recall (successfully identifying most of the actual positive examples). In summary, a high AUC value (whether for ROC or PRC) generally suggests strong classifier performance, reflecting its ability to make accurate and reliable predictions across different threshold settings.

Metrics for Regression

In regression tasks, model evaluation is based on quantitative metrics that measure the discrepancy between the predicted and the actual continuous target values. Let y_i denote the true value, $\hat{y}_i$ the predicted value for the i-th instance, and n the total number of samples. The most commonly used metrics include:

- Mean Squared Error (MSE), which penalizes larger deviations quadratically, emphasizes large errors, and is sensitive to outliers:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Root Mean Squared Error (RMSE), defined as the square root of the MSE, providing an error measure in the same units as the target variable:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

- Mean Absolute Error (MAE), which measures the average absolute difference between predicted and true values, and is more robust to outliers:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- Coefficient of Determination (R^2), which indicates the proportion of variance in the dependent variable explained by the model:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$$

4.4 Supervised Machine Learning Methods

Several well-established algorithms form the foundation of supervised machine learning, each employing a distinct inductive strategy to infer general patterns from labeled data and make predictions based on them. These methods typically operate on statistical correlations within feature-based representations and have proved to be successful across many domains.

Decision Tree Decision trees [2, 6–9] build a model in the form of a tree structure, where internal nodes represent tests on input features, and leaves correspond to output classes. They are intuitive and interpretable, however, they are also prone to overfitting, especially when the tree grows too deep. While they rely on statistical principles to partition the data, their learned structure can be expressed as a set of human-readable "if–then" rules, bridging statistical approaches, focused on pattern recognition from data, and symbolic approaches, centered on rule-based reasoning and explicit knowledge representation.

Ensemble Learning Ensemble learning [2] is based on the idea of combining the predictions of multiple models to enhance both robustness and predictive accuracy. By aggregating different learners, these methods mitigate overfitting and often outperform individual models. However, typically consisting of a large number of learners, ensembles are more difficult to interpret. As a result, it becomes challenging to understand which factors contributed to the improved performance or how individual predictions

are made [10]. Typical ensemble methods include Random Forests [11], composed of multiple decision trees, and boosting techniques. Boosting [12, 13] is an iterative method in which each new model is trained to correct the errors of the previous ones. Misclassified instances are given greater weight, forcing subsequent models to focus on the harder cases. The final prediction is obtained through a weighted vote (or averaging, in the case of regression), often resulting in significantly improved accuracy. AdaBoost and Gradient Boosting are widely used boosting algorithms.

Support Vector Machines Support Vector Machines (SVMs) [14] aim to find an optimal hyperplane that separates data points of different classes with the largest possible margin. They are particularly powerful in high-dimensional spaces and are well-suited for problems where the decision boundary is not linear, thanks to the use of kernel functions.

Neural Networks Neural networks [15, 16], especially deep architectures, model complex, non-linear relationships by composing multiple layers of interconnected units (neurons). Their flexibility and scalability have led to state-of-the-art results in areas such as image recognition, natural language processing, and speech analysis. However, they are often criticized for being black-boxes and their lack of interpretability. These methods learn from examples by optimizing numerical criteria (e.g., error minimization) and rely on inductive generalization to make predictions on new data. However, they often lack the ability to perform explicit reasoning, to incorporate prior knowledge in a structured way, or to generate interpretable models that align with human-understandable logic. This highlights a key distinction in ML: the contrast between sub-symbolic methods (which infer models through numeric optimization) and symbolic learning, which emphasizes structured representations and logical inference.

4.4.1 Limitations of statistical models

Despite their widespread success, purely statistical machine learning models face several limitations, particularly in terms of interpretability. These models often operate as black boxes, learning correlations from data without providing insight into the underlying decision-making process. This lack of transparency poses challenges in critical domains such as healthcare, law, and scientific discovery, where understanding why a

model makes a certain prediction is as important as the prediction itself. Moreover, statistical learning methods, such as most of those discussed in the previous section, typically represent input data as flat feature vectors and focus on learning statistical associations rather than explicit, interpretable rules. This representation makes it difficult to effectively handle structured or relational data that cannot be flattened without exponentially increasing the number of features. These limitations highlight the need for alternative approaches that can handle structured representations and support explicit reasoning, motivating the adoption of logic-based learning as a powerful alternative to standard ML techniques.

4.5 Overview of Large Language Models

In very recent years, Generative AI has emerged as a dominant technology in a variety of fields. With generative AI [17], we refer to those AI techniques that generate new content in the form of text, code, images, audio, and video. One type of Generative AI are Foundation Models (FMs) [18], very large ML models trained on enormous quantities of generic and unlabeled data and thus capable of performing a wide range of tasks. Among FMs, Large Language Models (LLMs) are based on the transformer architecture [19–21]. They are composed of multiple layers of neural networks, with an encoder and a decoder [22,23] that process sequences of text, capturing the connections and the relationships between words. Thanks to the self-attention mechanism [20], the model can consider distant tokens as well. To represent words, LLMs use multi-dimensional vectors, called word embeddings, in a way that keeps related words close to each other in the embedding space. These models often have billions of parameters and can be trained on vast amounts of data, enabling them to learn not only the grammar but also the semantics of a language. Given an input sequence, parameters are continuously adjusted during the training phase to maximize the likelihood of the next tokens.

LLMs have gained popularity in recent years [24] thanks to their ability to generate meaningful content from little input. Various LLMs have been specifically trained and used for automatic code generation, showing good performances [24, 25], such as OpenAI's Codex [26], and its GitHub implementation, Copilot. For example, authors of [27] have tested Codex to determine if it can identify security flaws and fix them, obtaining positive results on hand-crafted datasets; however, real-world scenarios proved

to be more challenging. As shown in [28], Copilot can become a useful tool for software developers for implementing simple procedures, speeding up their work. Given their expertise, even if the generated code contains bugs, they should be able to identify and fix them.

4.5.1 Prompt Engineering

Prompting is crucial when working with LLMs to get the most relevant and useful answers. A carefully crafted prompt can significantly impact the process by precisely defining the goal we aim to achieve. A basic prompt usually includes only the instructions for the task at hand, such as "summarize" or "classify". Other elements that can be included are: context; the audience; the role of the model; input data; output format; and examples of inputs and outputs [21]. The tone of the response can also be controlled. Additionally, the order also matters: information in the initial or in the final part of the prompt is often given more weight than that in the middle. This behavior is referred to, respectively, as primacy effect and recency effect [21, 29]. These components are not always mandatory, but the more specific the prompt, the better the response will be. Giving an overly general prompt to the LLM may result in its inability to complete the given task.

One or more examples of the desired output can also be included in the prompt. This is referred to as in-context learning or few-shot learning [24, 30], in which a few examples of the task at hand and a test example are provided, as shown in Example 4.5.1 taken from [24]. The authors of [21, 31] proved that implementing these strategies leads to improved performances.

Example 4.5.1. Few-shot prompting example from [24].

Prompt: A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.
To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:

Model answer: One day when I was playing tag with my little sister, she got really excited and she started doing these crazy farduddles.

Prompt chaining is a technique in which the output of one prompt is used as the input to the next, creating in this way a chain of prompts. This approach is particularly useful for complex tasks, where decomposing the problem into smaller parts can lead to improved performance and more interpretable intermediate results [32]. Incremental prompting, instead, involves providing a series of prompts to guide the model towards a more accurate and detailed response. By breaking the task into smaller steps, this technique helps the model better understand the request and reduces the errors that may arise from a single long prompt. The key difference with prompt chaining is that, in incremental prompting, new information is added with each step. Another prompting technique is Chain-of-Thought (CoT) Prompting, where the model generates a sequence of intermediate reasoning steps that lead from the given input to the final output. CoT is at the base of LLMs reasoning [33, 34], as shown by OpenAI's reasoning models [35]. Before generating an answer, these models learn to refine their thinking process, correct their mistakes, and explore different paths, building a chain of thought that leads to a higher-quality final result.

One of the main challenges in using LLMs is designing prompts that are both clear and effective in guiding the model toward the desired output. The quality of the prompt has a significant impact on the model's performance, and even small variations in wording or structure may lead to different results. Crafting an optimal prompt is therefore not straightforward, but is a gradual, iterative process that requires careful experimentation to determine what works best for each specific use case. Another major challenge is handling the inherent non-determinism of LLMs, as the same prompt can yield different outputs across multiple runs, making it difficult to ensure consistency and reproducibility.

Chapter 5

Logic Foundations

Logic provides a powerful and formal framework for representing structured knowledge and model reasoning. It is often called *formal* or *symbolic logic* because it adopts symbols. This section provides notions of logic and logic programming. First, it introduces propositional and first-order logic, which provide the formal basis for symbolic reasoning. It then presents logic programming, the foundation of most Inductive Logic Programming (ILP) systems. Finally, it covers the integration of logic and probability through probabilistic logic programming, enabling reasoning in uncertain or noisy domains. The notions in this chapter are based on [36, 37].

5.1 Propositional Logic

Propositional logic is the most basic formal system used in knowledge representation and reasoning. It operates on atomic propositions and their combinations through logical connectives, forming statements, also referred to as logical expressions, or formulas. The goal is to evaluate the truth value of these formulas under specific truth assignments, or to determine whether there exists an assignment that satisfies a given formula.

5.1.1 Syntax

The syntax of propositional logic defines the formal structure that determines which expressions are considered syntactically valid, and are called Well-Formed Formulas

p	q	p ∧ q	p ∨ q	¬p	p → q	p ↔ q
0	0	0	0	1	1	1
0	1	0	1	1	1	0
1	0	0	1	0	0	0
1	1	1	1	0	1	1

Table 5.1: An example of a truth table.

(WFFs). The syntax is based on a predefined alphabet, the set of symbols necessary to build propositions, which consists of the following:

- Atomic propositions: basic elements, typically denoted by symbols such as *a,p,q*.
- Logical connectives: ∧ (and), ∨ (or), ¬ (negation), → (implication), ↔ (equivalence), used to build compound propositions.
- Auxiliary symbols: left parenthesis "(" and right parenthesis ")", used to clarify the order of operations in complex expressions.

WFFs are built in a recursive manner: starting from atomic propositions, more complex formulas are generated by applying logical connectives and grouping subformulas using parentheses. Specifically:

5.1.2 Semantic

The semantics of propositional logic is defined by the assignment of a truth value to each atomic proposition of the formula, and thus to the formula itself. The truth value can be either true (denoted by 1, T, *true*, or ⊤) or false (denoted by 0, F, $false$, or ⊥). The assignment of truth values is called *interpretation*.

A WFF is said to be satisfiable if there exists an interpretation that satisfies it, i.e., that makes it true; unsatisfiable if it is false under all truth assignments; a tautology if it is true under all possible truth assignments.

Despite its simplicity, propositional logic has significant limitations, such as the fact that it cannot use objects or model relations and quantification, which are essential in most real-world domains. This motivates the use of first-order logic, which extends propositional logic with variables, quantifiers, and predicates.

5.2 First-Order Logic

First-Order Logic (FOL), also known as predicate logic, extends propositional logic by introducing variables and quantifiers. This allows FOL to provide a much more expressive formalism for representing structured domains involving objects, their properties, and relations, making it suitable for modeling a wide range of real-world scenarios.

5.2.1 Syntax

Like propositional logic, FOL has an alphabet consisting of the following symbols:

- Constants: domain objects, denoted by symbols starting with lowercase letters (e.g., *a, john*)
- Variables: placeholders referring to domain objects (e.g., *A, P, X*)
- Function symbols or functors: strings used to identify domain objects through a relationship between other n (called *arity*) domain objects (e.g., *father_of(bob)*).
- Predicate symbols: strings representing properties or relations of n objects, where n is the arity (e.g., *human(X), mother(mary, john)*)
- Logical connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$
- Quantifiers: universal quantifier $\forall$ ("for all X in the domain"), and existential quantifier $\exists$ ("there exists an X in the domain").

The scope of the quantifier is the formula that immediately follows it, and parentheses can be used to modify it. A variable is said to be *free* if it does not appear in the scope of any quantifier, and *bound* otherwise. For example, in the formula $\forall X p(X, Y)$, X is bound, and Y is free.

With the elements of the alphabet, one can define other constructs:

- Term: a variable, constant, or a function applied to terms
- Atom (or atomic formula): a predicate symbol applied to terms.
- Literal: an atom or its negation. A positive literal is an atom, while a negative literal is the negation of an atom.

In FOL, WFFs are syntactically correct formulas, built starting from atomic formulas combined with logical connectives and quantifiers. A WFF in FOL can be defined recursively:

- every atom is a WFF;
- if A and B are WFF, then so are $\neg A$, $A \wedge B$, $A \vee B$, $A \rightarrow B$ ($B \leftarrow A$), $A \leftrightarrow B$;
- if A is a WFF and X is a variable, $\forall X A$, and $\exists X A$ are WFFs.

A *clause* in FOL is a disjunction where all the variables are universally quantified:

$$\forall X \; h_1 \vee \ldots \vee h_n \vee \neg b_1 \vee \ldots \vee \neg b_m$$

where each h_is and b_js are atoms and X is the set of variables occurring in the clause. The universal quantifier is often implicit and thus omitted. The same clause can be written as:

$$h_1; \ldots; h_n \leftarrow b_1, \ldots, b_m$$

where semicolons represent disjunctions, while commas are used for conjunctions. The left part of the clause is the *head*, while the right part is the *body*. The clause can be read as "if the body is true, then the head is true". An example of a clause is $mother(X);\ father(X) \leftarrow parent(X)$, stating that if X is a parent, then X is either a mother or a father.

A clause that has no positive literal is a *denial*, if it has one positive literal is *definite*, and if it has more than one positive literal is called *disjunctive*. A *fact* is a definite clause without negative literals, and the body is omitted (e.g., *mother(jenny)*). A *Horn clause* is either a definite clause or a denial.

An *expression* is a literal, a term, or a clause.

A *substitution* is the process of replacing variables in the clause with constants. The application of a substitution $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ to an expression E results in a new expression $E\theta$, called an *instance* of E. If all variables have been replaced by constants, then E is said to be *ground*. The process of replacing variables with constants to make a clause or a program ground is called *grounding*.

The process of looking for a substitution that makes two logical expressions syntactically equal is called *Unification*.

A *theory* is a set of clauses. The *Herbrand universe* of a theory T is the set of all ground terms that can be generated by combining the constants and function symbols in T. The Herbrand universe is finite if T does not contain function symbols. Similarly, the *Herbrand base* is the set of all ground atoms that can be formed using the predicate symbols in T.

5.2.2 Semantics

The semantics of FOL defines how to assign a meaning to the formulas of a theory. In this process, each term is mapped to an object of the domain, and each formula is assigned a truth value. The semantics is based on the concept of *Herbrand interpretations* and *Herbrand models*.

A Herbrand interpretation (or *interpretation*) is a set of ground atoms and a subset of the Herbrand base. Given an interpretation I, it is possible to assign a truth value to formulas as follows:

- A ground atom $p(t_1, \ldots, t_n)$ is true in I iff $p(t_1, ..., t_n) \in I$.
- A conjunction of atoms $a_1, \ldots, a_n$ is true in I iff $\{a_1, \ldots, a_n\} \subseteq I$.
- A ground clause $h_1; \ldots; h_n \leftarrow b_1, \ldots, b_m$ is true in I iff at least one atom in the head is true when the body is true.
- A clause C is true in I iff every ground instance of C, obtained by substituting its variables with terms from the Herbrand universe, is true in I.
- A set of clauses Σ is true in I iff all clauses $C \in \Sigma$ are true in I.

A Herbrand model (or *model*) is an interpretation I that satisfies a set of clauses Σ ($I \models \Sigma$), that is, Σ is true in I. A set of clauses is satisfiable if it is satisfied by some interpretation, and unsatisfiable otherwise. If all models of a set of clauses Σ are also models of a clause C, then Σ logically entails C, or equivalently C is a logical consequence of Σ ($\Sigma \models C$).

5.3 Logic Programming

Logic is an effective approach to modeling domains with complex relationships among entities. Logic Programming (LP) [36] has been a powerful declarative paradigm based

on formal logic since the 70s [38]. Contrary to the traditional imperative programming paradigm, where the focus is on *how* the problem should be solved, LP falls under the declarative programming paradigm, where the program is a description of *what* the problem is. Logic programs represent knowledge of a domain using facts and rules to state what is true. A LP system can derive new knowledge by means of logical inference and answer users' queries about knowledge not explicitly encoded in the program. Since the inference procedure is independent of the knowledge base, logic programs are generally flexible, reusable, and easy to maintain and adapt as new requirements arise. New knowledge can be added without changing the inference algorithm, though it needs to be carefully structured. LP include languages based on formal logic, such as Prolog [39] , Datalog [40], and Answer Set Programming [41]. Syntax in LP is based on FOL, though it may slightly differ depending on the language.

Prolog

Prolog (Programmation en logique, or Programming in logic) was the first logic programming language, originally developed in 1972 for applications in AI involving symbolic computation and reasoning. In Prolog, knowledge is represented through relations expressed as Horn clauses, which define logical rules and facts. Example 5.3.1 illustrates a simple Prolog program.

Example 5.3.1. Logic program in Prolog encoding a graph reachability problem. The `edge/2` predicate represents an edge between two nodes, while the `path/2` predicate defines reachability: there is a path from a node to itself, and a path from a node `X` to a node `Y` if there exists a node `Z` such that there is an edge from `X` to `Z` and a path from `Z` to `Y`.

```
edge(a,b).
edge(a,c).
edge(b,c).

path(X,X).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

A query asks whether a given goal can be logically derived from these rules. To answer queries, Prolog employs the SLD (Selective Linear Definite) [42] resolution [43].

Starting from a goal or query, represented as a conjunction of literals,

$$? - l_1, \ldots, l_j$$

a literal, for example l_1, is iteratively selected and replaced with the body of a rule $h : -b_1, ..., b_m$ such that h unifies with l_1 with substitution θ. The new goal then becomes

$$? - (b_1, ..., b_m, l_2, ldots, l_j)\theta$$

This process generates an SLD-tree, where each node represents a goal, and each edge corresponds to a resolution step. A path from the root to a leaf constitutes an SLD derivation. If an empty goal is eventually derived, the query succeeds; otherwise, if no applicable rules remain, the query fails. Prolog selects the leftmost literal and adopts a depth-first search strategy with chronological backtracking. Prolog operates under the closed world assumption, meaning that any statement that cannot be proven from the program is assumed to be false.

Answer Set Programming

Answer Set Programming is another example of LP language and is an expressive formalism used for modeling combinatorial domains. An answer set program consists of a collection of normal rules of the form

$$h : -b_0, \ldots, b_m,\ not\ c_0, \ldots,\ not\ c_n$$

where h, the b_is, and the c_is are *atoms*. A rule without the head is called *constraint*. ASP also introduces choice rules, which allow the non-deterministic selection of a subset of atoms satisfying certain conditions, and aggregates, which enable numerical or set-based constraints.

5.3.1 Semantics

The semantics of a logic program P defines which atoms a are entailed by the program, $P \models a$. Semantics for logic programs without negation can be defined in terms of Herbrand models. Given a set of definite clauses P, the intersection of all the Herbrand models of P is a model itself, and is referred to as the minimal or least Herbrand model,

written $lhm(P)$. This model is unique, minimal, and always exists. It serves as the basis for the *model-theoretic semantics* of the program, which is the set of all the ground atoms that are logical consequences of P: $P \models a$ iff $a \in lhm(P)$, where a is a ground atom.

In a normal program, clauses may contain negative literals in the body and thus take the form:

$$h : -b_1, \ldots, b_n, \neg c_l, \ldots, \neg c_m$$

where h, b_i, c_j are atoms. The presence of negated atoms introduces non-monotonicity, since adding new knowledge can invalidate previous conclusions. Different approaches have been developed to deal with non-monotonic reasoning, such as the Stable Model semantics [44] or the Well-Founded semantics [45].

5.4 Probabilistic Logic Programming

Probabilistic Logic Programming (PLP) combines uncertainty and logic-based languages [37]. Given its expressiveness, in the last decades PLP, and in particular PLP under the Distribution Semantics [46], has been widely adopted in domains characterized by uncertainty [47–51].

5.4.1 Distribution Semantics

The Distribution Semantics (DS) is one approach for combining logic and probabilities. Under the DS, a probabilistic logic program without function symbols defines a probability distribution over a set of normal logic programs, also called *instances* or *worlds*. Uncertainty is modeled by associating probabilistic choices with the clauses of a program. Every random choice defines an alternative version of a clause, and the set of choices, typically assumed to be mutually independent, is associated with a probability distribution. The distribution is extended to a joint distribution over worlds and interpretations (or queries). The probability of a query is the sum of the probabilities of the worlds where the query is true [37]:

$$P(Q) = \sum_w P(Q, w) = \sum_w P(Q|w)P(w) = \sum_{w \models Q} P(w),$$

where $P(Q|w) = 1$ if Q is true in w and 0 otherwise.

Probabilistic Logic Programs [52], Logic Programs with Annotated Disjunctions (LPADs) [53], PRISM [54], and ProbLog [55] are languages under the DS.

ProbLog A ProbLog [55] program consists of a set of (certain) rules and a set of probabilistic facts of the form $p_i : a_i$, where $p_i \in [0, 1]$ and a_i is an atom, meaning that each ground instantiation of a_i is true with probability p_i and false with probability $1 - p_i$. In this setting, the probability distribution is defined over facts. Each probabilistic fact represents an independent random choice, and the corresponding possible worlds are generated by deciding, for each grounded probabilistic fact, whether it is included or excluded from the program.

Example 5.4.1. ProbLog program stating that a person sneezes if they have the flu and the flu is the active cause of sneezing, or if they have hay fever and hay fever is the active cause of sneezing. There are 4 possible worlds: $w_1 = \{\text{flu\_sneezing(bob)}, \text{hay\_fever\_sneezing(bob)}\}$, $w_2 = \{\text{flu\_sneezing(bob)}\}$, $w_3 = \{\text{hay\_fever\_sneezing(bob)}\}$, and $w_4 = \{\}$. The probabilities of each world are: $P(w_1) = 0.7 \times 0.8$, $P(w_2) = 0.3 \times 0.8$, $P(w_3) = 0.7 \times 0.2$, and $P(w_4) = 0.3 \times 0.2$. For the query $sneezing(bob)$, which is true in 3 worlds, the probability $P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$.

```
sneezing(X) :- flu(X), flu_sneezing(X).
sneezing(X) :- hay_fever(X), hay_fever_sneezing(X).
flu(bob).
hay_fever(bob).
0.7 :: flu_sneezing(X).
0.8 :: hay_fever_sneezing(X).
```

LPAD LPADs [56] are another PLP language under the DS. In LPADs without function symbols, the head of each clause is a disjunction of atoms, where each atom is annotated with an associated probability. These probabilities specify a distribution over the possible head atoms of a clause, indicating which atom is selected when the clause is instantiated. A special atom, typically denoted as `null`, can be used to represent the case where none of the head atoms is chosen. However, `null` does not appear in the body of any rule and serves only to complete the probability distribution. Possible worlds are obtained by grounding the program and then selecting one head

atom for each grounding of every probabilistic clause, according to the corresponding probability distribution. Each complete selection defines a world, and the probability of that world is computed as the product of the probabilities of the atoms chosen in that world.

Example 5.4.2. LPAD encoding the same scenario of Example 5.4.1. We want to compute the probability of the query $sneezing(bob)$. There are 4 possible worlds, each including also the facts $flu(bob)$ and $hay\_fever(bob)$: $w_1 = \{sneezing(X) : -flu(bob), sneezing(bob) : -hay\_fever(bob)\}$, $w_2 = \{null : -flu(bob), sneezing(bob) : -hay\_fever(bob)\}$, $w_3 = \{sneezing(X) : -flu(bob), null : -hay\_fever(bob)\}$, and $w_4 = \{null : -flu(bob), null : -hayfever(bob)\}$. The probabilities of each world are: $P(w_1) = 0.7 \times 0.8$, $P(w_2) = 0.3 \times 0.8$, $P(w_3) = 0.7 \times 0.2$, and $P(w_4) = 0.3 \times 0.2$. Since $sneezing(bob)$ is true in 3 worlds, the probability $P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$.

```
sneezing(X) : 0.7 ; null : 0.3 :- flu(X).
sneezing(X) : 0.8 ; null : 0.2 :- hay_fever(X).
flu(bob).
hay_fever(bob).
```

5.5 A hybrid approach: NeSy

Neural-symbolic (NeSy) AI [57–60], is an emerging field that focuses on integrating deep learning with symbolic knowledge, combining the strengths of both paradigms: the powerful learning capabilities of neural networks and the expressive, structured representations offered by symbolic AI. This integration also helps mitigate their respective limitations. Neural networks, despite their effectiveness, have limited reasoning abilities and are often regarded as black-box models, making their internal processes difficult to interpret and their output difficult to explain. Symbolic systems, on the other hand, struggle with real-world complexity and to adapt effectively to novel or evolving scenarios.

A well-known example of NeSy system is DeepProbLog [59], which extends ProbLog by adding neural predicates. A DeepProbLog program is a ProbLog program extended

with a set of ground neural annotated disjunctions of the form

$$nn(m_q, \vec{t}, \vec{u}) :: (q, \vec{t}, u_1); \ldots; (q, \vec{t}, u_n) : -b_1, \ldots, b_m$$

where the b_i are body atoms, $\vec{t} = t_1, \ldots, t_k$ is a vector of ground terms representing the inputs of the neural network for predicate q, while $u_1, \ldots, u_n$ are possible output values of the neural network. m_q indicates a neural network that specifies a probability distribution over its output values, given an input.

As an example, consider the predicate `addition(X, Y, Z)`, where `X` and `Y` are images of digits and `Z` is their sum. In this case, the neural network recognizes the digits in the images, and the resulting classifications can then be used as arguments of the logical predicate `addition/3`.

Chapter 6

Inductive Logic Programming

ILP lies at the intersection of ML and LP, combining the strengths of both paradigms. It offers two main advantages: the ability to represent and reason over relational data, and the inherent explainability of logic-based models. The hypotheses learned by ILP are expressed as symbolic rules, which are naturally interpretable even by non-experts, enhancing trust and a deeper understanding of the model's behavior. This makes ILP a natural framework for explainable AI, particularly suitable for those domains where transparency and accountability are critical.

An ILP task is composed of three sets: a set of positive examples E^+, a set of negative examples E^-, and a background knowledge B, which contains rules whose truth is known and represents the knowledge of the domain. The goal is to induce a hypothesis that generalizes training examples, using logic programs to represent data. ILP usually works under the closed-world assumption [61], meaning that everything that is not included in the program is considered false.

6.1 Learning Setting

Formally, the definition of an ILP task depends on how the input and the output are represented.

6.1.1 Learning from entailment

Given a set of positive examples E^+, a set of negative examples E^-, a background knowledge B, and a space of possible programs $\mathcal{H}$, the goal is to find a program $P \in \mathcal{H}$

such that:

- $\forall e^+ \in E^+,\ P \cup B \models e^+$ (completeness)
- $\forall e^- \in E^-,\ P \cup B \not\models e^-$ (consistency)

Examples of ILP systems that use this setting are FOIL [62] and Progol [63].

6.1.2 Learning from interpretations

In the LFI setting, positive and negative examples are given as a set of complete or partial interpretations, typically represented as a set of ground facts. Given a set of positive interpretations P, a set of negative interpretations N, and a space of possible clausal theories $\mathcal{H}$, the goal is to find a clausal theory $H \in \mathcal{H}$ such that:

- $\forall p \in P, p \models H$
- $\forall n \in N, n \not\models H$

An example of a system working in this setting is ICL [64].

6.2 Hypothesis Space

An ILP system searches for a suitable hypothesis within a hypothesis space, which encompasses all possible programs that can be constructed using the chosen representation language. Since this space is typically infinite, one of the key challenges in ILP is how to effectively restrict it so that the search process becomes computationally feasible.

6.2.1 Language Bias

The language bias determines the structure of the hypothesis space by imposing constraints on the clauses that can be learned, in terms of the predicates and their arguments.

One of the most common forms of language bias is the use of mode declarations [63]. These declarations specify which atoms may appear in the head and the body of rules, how many times they can occur, and the types of their arguments. In addition, they

define whether an argument corresponds to an input variable, an output variable (introduced in the head or a preceding literal), or a constant. Consider the snippet below:

```
modeh(*,pos).
modeb(*,triangle(-obj)).
modeb(*,square(-obj)).
modeb(*,circle(-obj)).
modeb(*,inside(+obj,-obj)).
modeb(*,inside(-obj,+obj)).
modeb(*,config(+obj,-#dir)).
```

Here, `modeh` specifies which atoms can appear in the head of a rule, while `modeb` defines which atoms may be included in the body. The first argument of each mode declaration indicates its recall, that is, how many times the declaration can be used within a single rule. The recall can either be an integer value or the symbol *, which denotes no limit. The symbols +, -, and # indicate input variables, output variables, and constants, respectively.

Choosing appropriate mode declarations is a time-consuming yet crucial process, so that the system performs well, as it has a significant impact on system performance. Overly general or weak mode declarations may lead to poor generalization and inefficient search, whereas excessively restrictive ones can prevent the system from discovering the correct hypothesis [65].

6.2.2 Search Strategy

The search strategy defines how the hypothesis space is explored in order to identify a suitable hypothesis. The organization of the search space is typically based on a generality ordering, which provides a structured way to explore the hypothesis space by comparing the relative generality of hypotheses. Given two hypotheses h_1 and h_2, h_1 is more general than h_2 if h_1 entails at least as many examples as h_2. Conversely, h_1 is more specific than h_2 if h_1 entails fewer examples than h_2.

One of the most widely adopted generality relations in ILP is θ-subsumption [66]. Given two clauses (sets of literals) C_1 and C_2, C_1 θ-subsumes C_2 (denoted $C_1 \geq C_2$) if there exists a substitution θ such that $C_1\theta \subseteq C_2$. If $C_1 \geq C_2$ then $C_1 \models C_2$, meaning that C_1 is more general than C_2. The opposite does not necessarily hold.

θ-subsumption is generally preferred to entailment, since entailment is often computationally expensive.

In ILP, finding the correct hypothesis is typically performed with generalization and specialization. If the current hypothesis H, together with the background knowledge B is not complete (it fails to cover some positive examples), then the learner must find a more general hypothesis that covers all positive examples while remaining consistent with the negative ones. On the other hand, if H, combined with B, is not consistent (it incorrectly covers some negative examples), the learner looks for a more specific hypothesis that avoids covering negative examples. ILP systems implement this process by iteratively modifying the current hypothesis and evaluating it against the available examples and background knowledge using a refinement operator. Specialization operators add literals, unify variables, or ground variables, while generalization operators perform the opposite operations. By applying these refinement operators, ILP systems traverse the hypothesis space and incrementally refine candidate hypotheses until a suitable solution is found.

Search strategies in ILP are commonly categorized as either top-down or bottom-up. In a top-down approach, the search begins with an overly general hypothesis and iteratively specializes it in order to exclude negative examples. In contrast, a bottom-up approach starts from an overly specific hypothesis and gradually generalizes it so as to cover more positive examples. Each approach offers different advantages depending on the problem domain and the background knowledge available. Top-down methods tend to be more guided by the structure of the hypothesis language, whereas bottom-up methods often rely more heavily on the examples themselves.

6.3 Probabilistic Inductive Logic Programming

Probabilistic Inductive Logic Programming (PILP) [67] refers to the integration of probabilistic reasoning and ILP, aiming to combine the expressive power of first-order logic with the ability to model uncertainty. Traditional ILP systems assume that all facts and examples are certain and deterministic; however, in real-world domains data is often incomplete, noisy, or ambiguous. PILP frameworks address this limitation by associating probabilities with logical statements, thus enabling reasoning and learning in uncertain settings. Learning in these frameworks typically involves estimating the structure, i.e., the rules (*structure learning*) and the associated probabilities (*parameter*

learning) that best explain the observed data.

6.3.1 Parameter Learning

Formally, the problem of parameter estimation can be stated as follows: given a probabilistic logic program and a set of observed examples, the goal is to estimate the values of the parameters that best explain the data.

One algorithm for parameter learning is the Expectation-Maximization (EM) algorithm. EM is an iterative method used to estimate model parameters in the presence of latent (unobserved) variables. In the Expectation (E) step, the algorithm computes the expected distribution of the unobserved variables given the current parameter estimates and the observed data. In the Maximization (M) step, the parameters are re-estimated using these expectations, typically by maximizing the expected log-likelihood, often through relative frequency estimation. The process alternates between the E and M steps until convergence, i.e., until the likelihood of the data no longer improves significantly. Alternatively, gradient-based optimization methods, such as gradient descent, can be used for parameter learning. These methods iteratively adjust the model parameters in the direction of the negative gradient of a loss (or likelihood) function, seeking to minimize it. Gradient descent provides a more general framework, though it may converge only to a local optimum. LFI-ProbLog [68] and EMBLEM [69] are examples of parameter learning algorithms.

EMBLEM

EMBLEM (Expectation Maximization over Binary Decision Diagrams for Probabilistic Logic Programs) [69], is a parameter learning algorithm that estimates the parameters of general LPADs from data using the Expectation–Maximization algorithm.

Given an LPAD P with unknown parameters and two sets $E^+ = e_1, ..., e_k$ and $E^- = e_{k+1}, ..., e_n$ of positive and negative examples (ground atoms), EMBLEM learns the parameters Π of P that maximize the likelihood of the examples:

$$argmax_{\Pi} P(E^+, E^-) = argmax_{\Pi} \prod_{i=1}^{k} P(e_i) \prod_{i=k+1}^{n} P(\neg e_i).$$

An LPAD consists of a set of annotated rules with parameters, representing general

knowledge about the domain, and a set of certain ground facts, representing background information about individual cases within a specific world. It may be useful to provide information corresponding to multiple worlds; therefore, for each world, a background knowledge, a set of positive examples, and a set of negative examples are provided. The description of a single world is often referred to as a mega-interpretation or mega-example. The goal is then becomes the maximization of the joint likelihood of all examples across the different mega-interpretations, i.e., the product of their individual likelihoods.

Thus, EMBLEM takes as input a set of goals representing the examples, and for each goal it generates a Binary Decision Diagram (BDD) encoding all its possible explanations. The examples are grouped into interpretations, sets of ground facts that describe distinct portions of the domain of interest. Queries correspond to ground atoms whose predicates have been defined by the user as target predicates. EMBLEM computes the expectations required by EM via knowledge compilation: for each example, it builds a BDD encoding all its explanations. During learning, EMBLEM iteratively computes the expectations of the hidden variables directly over the BDDs, as well as the likelihood; then, it updates the parameters by relative frequency. The EM cycle repeats until convergence.

6.3.2 Structure Learning

Parameter learning can be performed when the structure of the program is known. However, in many real-world domains, this is not the case, and both the structure and the parameters of the program must be learned directly from data. This task, known as structure learning in PILP. ProbFOIL [70], ProbFOIL+ [71], and SLIPCOVER [72] are examples of structure learning systems.

SLIPCOVER

SLIPCOVER (Structure LearnIng of Probabilistic logic programs by searChing OVER the clause space) [72] learns both the structure and the parameters of general LPADs from data.

Similarly to what EMBLEM does, SLIPCOVER takes as input a set of mega-examples and an indication of the target predicates, i.e., those we want to predict. Each mega-example provides both positive and negative examples for all predicates that may

appear in the head of clauses, either target or non-target (background predicates). Starting from an empty or trivial LPAD and a set of interpretations, SLIPCOVER finds the model and the parameters that maximize the log-likelihood of the data. The algorithm operates in two main phases. First, it performs a beam search over the space of probabilistic clauses to identify promising candidates. It is divided in two steps: the construction of a set of beams containing bottom clauses, and then a beam search over each beam to refine the bottom clauses. The bottom clause [63], denoted with $\bot$, is the most specific clause covering an example. The search over the space of possible clauses is guided by a language bias specified through mode declarations, which constrain the form and content of possible clauses. The second phase is a greedy search over the space of probabilistic programs, guided by the log-likelihood. Finally, parameter learning is done with EMBLEM.

Part III

Probabilistic Learning Systems

Chapter 7

LIFTCOVER+

Lifted inference [73] was introduced to improve the performance of reasoning in probabilistic relational models by taking into consideration populations of individuals instead of considering each individual separately. In this chapter, we introduce LIFTCOVER+, an ILP system that performs both parameter and structure learning of liftable probabilistic logic programs.

7.1 Setting

We focus on the *liftable PLP language* [74], which represents a subset of probabilistic logic programs for which inference can be performed in a lifted way. Within this framework, programs are defined as sets of clauses, each having the same single annotated predicate in the head. Formally, the clauses can be written as:

$$C_i = h_i : \pi_i :- b_{i1}, \ldots, b_{iu_i}$$

where the head atom is defined over the predicate *target/a*, with a being its arity. The clause bodies include predicates other than *target/a*, defined by facts and rules that are certain, having a single head atom with probability 1. The predicate *target/a* serves as the target of learning, while the remaining predicates are treated as *input predicates*. Thus, in the liftable PLP language, uncertainty appears only in the rules. The objective is to compute the probability of a query q of *target/a*. This is achieved by identifying all ground instantiations of clauses for *target/a* where the body is true and the head corresponds to q. Let $\{\theta_{i1}, ..., \theta_{im_i}\}$ be the m_i substitutions of such instantiations of

clause C_i, $i = 1, ..., n$. Every instantiation C_i corresponds to a random variable X_{ij} that equals 1 with probability π_i, and 0 with probability $1 - \pi_i$. The query q is true if at least one of the random variables associated with the rules is true, i.e., takes value 1. Conversely, the query q is false only when all random variables are false. Because these random variables are mutually independent, the probability of q being true can be computed as $P(q) = 1 - \prod_{i=1}^{n}(1 - \pi_i)^{m_i}$.

LIFTCOVER [74] learns the structure of liftable probabilistic logic programs. Given a set $E^+ = \{e_1, \ldots, e_Q\}$ of positive examples, a set $E^- = \{e_{Q+1}, \ldots, e_R\}$ of negative examples, and a background knowledge B (possibly a normal logic program defining the input predicates), the goal of structure learning is to identify a liftable probabilistic logic program T that maximizes the likelihood:

$$L = \prod_{q=1}^{Q} P(e_q) \prod_{r=Q+1}^{R} P(\neg e_r)$$

LIFTCOVER addresses this problem by first identifying suitable clauses through a top-down beam search guided by the log-likelihood (LL) of the data. During the search, the refinement operator extends the current clause by adding a literal selected from a bottom clause to its body. The beam search is repeated either a user-specified number of times or until the beam becomes empty. Parameter learning is then performed on the complete set of discovered clauses, treated as a single theory, either with Expectation-Maximization (EM) or Limited-memory BFGS (LBFGS). In particular, LBFGS estimates the parameters that maximize the likelihood using the gradient of the log-likelihood with respect to the parameters. The likelihood can be unfolded to

$$L = \prod_{l=1}^{n}(1 - \pi_l)^{m_{l-}} \prod_{q=1}^{Q}\left(1 - \prod_{l=1}^{n}(1 - \pi_l)^{m_{lq}}\right)$$

where m_{iq} (m_{ir}) is the number of instantiations of C_i whose head is e_q (e_r) and whose body is true, and $m_{l-} = \sum_{r=Q+1}^{R} m_{lr}$. As in [74], its gradient can be computed as:

$$\frac{\partial L}{\partial \pi_i} = \frac{L}{1 - \pi_i}\left(\sum_{q=1}^{Q} m_{iq}\left(\frac{1}{P(e_q)} - 1\right) - m_{i-}\right) \tag{7.1}$$

The equation $\frac{\partial L}{\partial \pi_i} = 0$ does not admit a closed-form solution, therefore, optimization is

needed to find the maximum of L. The clauses with a probability below a user-defined threshold are discarded.

In models with hidden variables, the EM algorithm [75] is employed to obtain the maximum likelihood estimates of parameters. During the Expectation step, the distribution of the latent variables in each instance is computed, given the observed data and the parameter values. During the Maximization step, the parameters are updated so as to maximize the expected likelihood. These two steps are iterated alternately until the likelihood converges, i.e., no further improvement is observed. Following [74], applying the EM algorithm requires computing the distribution of the hidden variables given the observed ones, namely $P(X_{ij} = 1|e)$ and $P(X_{ij} = 1|\neg e)$. Given that $P(X_{ij} = 1, e) = P(e|X_{ij} = 1) \cdot P(X_{ij} = 1) = P(X_{ij} = 1) = \pi_i$ and $P(e|X_{ij} = 1) = 1$,

$$P(X_{ij} = 1|e) = \frac{P(X_{ij} = 1, e)}{P(e)} = \frac{\pi_i}{1 - \prod_{i=1}^{n}(1 - \pi_i)^{m_i}} \tag{7.2}$$

$$P(X_{ij} = 0|e) = 1 - \frac{\pi_i}{1 - \prod_{i=1}^{n}(1 - \pi_i)^{m_i}} \tag{7.3}$$

Since $P(X_{ij} = 1, \neg e) = P(\neg e|X_{ij} = 1) \cdot P(X_{ij} = 1) = 0$ and $P(\neg e|X_{ij} = 1) = 0$,

$$P(X_{ij} = 1|\neg e) = 0 \tag{7.4}$$

$$P(X_{ij} = 0|\neg e) = 1 \tag{7.5}$$

7.2 Implementation

LIFTCOVER may induce large clause sets that tend to overfit the training data. To mitigate this, we introduce LIFTCOVER+, a modified version of LIFTCOVER, shown in Algorithm 1, that integrates regularization into the parameter learning process and optimizes the likelihood via EM or gradient descent.

Regularization is a widely used technique to prevent overfitting, in which a penalty term is added to the loss function to penalize large weights. This encourages the learning of a small number of clauses with large weights, while clauses with small weights, i.e., those having little impact on the probability of the query, can be discarded, thereby simplifying the theory. Although regularization is commonly applied in gradient-based

algorithms, it can be incorporated into the EM algorithm during the Maximization step, where the parameters that maximized the LL are estimated. In the EM algorithm, regularization can be Bayesian, L1, or L2.

In Bayesian regularization, the parameters are updated under the assumption of a prior distribution, typically modeled as a Dirichlet probability density with parameters $[a, b]$. This is equivalent to observing a additional occurrences of $X_{ij} = 1$ and b additional occurrences of $X_{ij} = 0$. When b is significantly larger than a, this has the effect of shrinking the parameters. The distinction between L1 and L2 regularization lies in the form of the penalty applied to the loss function: L1 adds the sum of the absolute values of the parameters to the loss function, whereas L2 adds the sum of their squares. Both approaches help to control overfitting by discouraging excessively large parameter values.

The L1 objective function [76] is:

$$J_1(\theta) = N_1 \cdot log\theta + N_0 \cdot log(1-\theta) - \gamma\theta \tag{7.6}$$

where $\theta = \pi_i$, N_0 and N_1 are the expected occurrences of $X_{ij} = 0$ and $X_{ij} = 1$ computed in the Expectation step, and γ is the regularization coefficient. The value of θ that maximizes J_1 is computed in the Maximization step by solving the equation $\frac{\partial J(\theta)}{\partial \theta} = 0$ [76]. $J_1(\theta)$ is maximum at

$$\theta_1 = \frac{4N_1}{2(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})} \tag{7.7}$$

The L2 objective function [76] is:

$$J_2(\theta) = N_1 \cdot log\theta + N_0 \cdot log(1-\theta) - \frac{\gamma}{2}\theta^2 \tag{7.8}$$

and value of θ that maximizes J_2, is:

$$\theta_2 = \frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3} - \frac{2\pi}{3}\right)}{3} + \frac{1}{3} \tag{7.9}$$

In LIFTCOVER+, LBFGS is replaced with regularized gradient descent. The

objective function is defined as the sum of the cross-entropy errors err_i across all examples:

$$err = \sum_{i=1}^{Q+R}(-y_i logP(e_i) - (1-y_i)log(1-P(e_i))) \tag{7.10}$$

where $Q+R$ is the total number of examples, e_i is an example, and y_i is its sign, thus y_i equals to 1 (0) if the example is positive (negative). L1 or L2 regularization can then be applied to minimize the loss function [76]:

$$err_{L1} = \sum_{i=1}^{Q+R} -y_i logP(e_i) - (1-y_i)log(1-P(e_i)) + \gamma \sum_{i=1}^{k} |\pi_i| \tag{7.11}$$

$$err_{L2} = \sum_{i=1}^{Q+R} -y_i logP(e_i) - (1-y_i)log(1-P(e_i)) + \frac{\gamma}{2} \sum_{i=1}^{k} \pi_i^2 \tag{7.12}$$

where k is the number of parameters, the π_is are the probabilities of the clauses, and γ is the regularization coefficient.

After parameter learning, any clauses with a probability below a predefined threshold are removed.

7.3 Experiments

The primary objective of the experiments is to evaluate whether incorporating regularization into LIFTCOVER+ enhances the quality of the learned solutions. All experiments were carried out on a GNU/Linux machine equipped with an Intel Core i3-10320 Quad Core 3.80 GHz CPU.

LIFTCOVER+ was evaluated on a total of 12 real-world datasets:

- UW-CSE [77] describes the Computer Science Department of the University of Washington. It is used to predict whether a student is advised by a particular professor.

- Mondial [78] contains information about the world's geographical regions, including population size, political system, and country border relationships.

Algorithm 1 Function LIFTCOVER

```
1: function LIFTCOVER(NB, NI, NInt, NS, NA, NV)
2:     Beam =InitialBeam(NInt, NS, NA)                ▷ Bottom clauses building
3:     CC ← ∅
4:     Steps ← 1
5:     NewBeam ← []
6:     repeat
7:         Remove the first couple ((Cl, Literals), LL) from Beam ▷ Remove the first clause
8:         Refs ←ClauseRefinements((Cl, Literals, NV))   ▷ Find all refinements Refs of (Cl, Literals)
9:         for all (Cl', Literals') ∈ Refs do
10:            (LL'', {Cl''}) ←LearnWeights(I, {Cl'})
11:            NewBeam ←Insert((Cl'', Literals'), LL'', NewBeam, NB)   ▷ The refinement is inserted in the beam in order of likelihood, possibly removing the last clause if the size of the beam NB is exceeded
12:            CC ← CC ∪ {Cl'}
13:        end for
14:        Beam ← NewBeam
15:        Steps ← Steps + 1
16:    until Steps > NI or Beam is empty
17:    (LL, Th) ←LearnWeights(CC)
18:    Remove from Th the clauses with a weight smaller than WMin
19:    return Th
20: end function
```

- Carcinogenesis [79] is a classic ILP benchmark dataset for Quantitative Structure-Activity Relationship (QSAR)studies, aiming to predict the carcinogenicity of chemical compounds based on their molecular structure and physicochemical properties.

- Mutagenesis [80] is another QSAR-focused ILP benchmark, where the goal is to predict the mutagenicity of compounds, a property related to carcinogenicity, from their chemical structure.

- Bupa[1] is used for diagnosing patients with liver disorders.

- Nba[1] is employed to predict the outcomes of NBA basketball games.

- Pyrimidine and Triazine[1] are QSAR datasets for predicting the inhibition of dihydrofolate reductase by pyrimidines and triazines, respectively.

- Financial [81][2] aims to predict the success of loan applications submitted by bank clients.

- Sisya and Sisyb, from [81][2], concern clients in the insurance sector, and are used to classify households and individuals with respect to private life insurance.

- Yeast [81][2] is used to predict whether a yeast gene encodes a protein involved in metabolism.

Table 7.1 summarizes the key characteristics of these datasets.

We compared four different configurations of LIFTCOVER+: EM with Bayesian regularization (EM-Bayes), EM with L1 regularization (EM-L1), EM with L2 regularization (EM-L2), and gradient descent with fixed learning rate $\eta = 0.0001$ and L1 regularization (GD). Hyperparameters for Bayesian regularization were set as $a = 0$ and b equal to 15% of the total number of examples in the dataset. We set $\gamma = 50$ for L1 and L2 in EM, and $\gamma = 10$ for L1 in gradient descent.

The parameters controlling structure learning, whose values are listed in Table 7.2, are the following:

- *NInt* is the number of mega-examples on which to build the bottom clauses.

- *NA* is the number of bottom clauses to be built for each mega-example.

- *NS* is the number of saturation steps (for building the bottom clauses).

- *NI* is the maximum number of clause search iterations.

- *NB* is the size of the beam.

- *NV* is the maximum number of variables in a rule.

- *WMin* is the minimum probability under which the rule is removed.

[1] https://relational.fit.cvut.cz/
[2] https://dtai.cs.kuleuven.be/ACE/doc/

Table 7.1: Characteristics of the datasets for the experiments: number of predicates (P), of tuples (T) (i.e., ground atoms), of positive (PEx) and negative (NEx) examples for target predicate(s), of folds (F). The number of tuples includes the target positive examples.

Dataset	P	T	PEx	NEx	F
Financial	9	92658	34	223	10
Bupa	12	2781	145	200	5
Mondial	11	10985	572	616	5
Mutagen.	20	15249	125	126	10
Sisyb	9	354507	3705	9229	10
Sisya	9	358839	10723	6544	10
Pyrimidine	29	2037	20	20	4
Yeast	12	53988	1299	5456	10
Nba	4	1218	15	15	5
Triazine	62	10079	20	20	4
UW-CSE	15	2673	113	20680	5
Carcinogen.	36	24533	182	155	1

Their values were set consistently with the original LIFTCOVER implementation [74], with the goal of ensuring a fair comparison between the two approaches.

All the configurations were evaluated using the Area Under the Precision-Recall Curve (AUC-PR) and the Area Under the Receiver Operating Characteristics Curve (AUC-ROC). Both metrics were calculated following the procedures described in [82, 83]. LIFTCOVER+ was compared with LIFTCOVER-EM from [74].

Tables 7.3, 7.4, and 7.5 show the performance of the different configurations in terms of average over the folds of AUC-ROC, AUC-PR, and the execution times, respectively. The results of LIFTCOVER-EM were taken from [74]. To enable a fair comparison, the execution times of LIFTCOVER+ were scaled by a factor of 3.8/2.4 to match the LIFTCOVER-EM measurements reported in [74], which were executed on a different machine, equipped with an Intel Xeon Haswell E5-2630 v3 (2.40GHz) CPU. Figure 7.1 shows the histograms of the above-mentioned data.

LIFTCOVER+ shows a slight improvement over LIFTCOVER-EM in terms of AUC-PR on several datasets: 6 out of 12 with EM-Bayesian and EM-L1, 7 datasets

Table 7.2: Parameters controlling structure search for LIFTCOVER+.

Dataset	*NB*	*NI*	*NInt*	*NS*	*NA*	*NV*	*WMin*
Financial	100	20	16	1	1	4	1e-4
Bupa	100	20	4	1	1	4	1e-4
Mondial	1000	10	1	2	6	5	1e-4
Mutagen.	100	10	4	1	1	4	1e-4
Sisyb	100	20	10	1	1	50	1e-4
Sisya	100	20	4	1	1	4	1e-4
Pyrimidine	100	20	4	1	1	100	1e-4
Yeast	100	20	12	1	1	4	1e-4
Nba	100	20	4	1	1	100	1e-4
Triazine	100	20	4	1	1	4	1e-4
UW-CSE	20	60	4	1	4	4	1e-4
Carcinogen.	100	60	16	2	1	3	1e-4

with EM-L2, and 3 datasets when using gradient descent (GD). Overall, the average AUC-PR across all datasets is highest for LIFTCOVER+ with EM and L2 regularization, followed closely by Bayesian regularization. On the other hand, results obtained with LIFTCOVER+ using GD were notably worse for the Pyrimidine and Yeast datasets and lower in most other cases. LIFTCOVER+ achieved a substantial improvement on the NBA dataset, reaching an AUC-PR over 0.7 compared to 0.55 obtained by LIFTCOVER-EM. However, the Sisyb dataset remains challenging for LIFTCOVER+, regardless of whether EM or GD is used. Regarding AUC-ROC, LIFTCOVER+ outperforms LIFTCOVER-EM on 4 datasets with EM-Bayes and EM-L1, on 3 datasets with EM-L2, and on 2 datasets with GD. In general, GD tends to degrade performance in most cases, likely due to the highly non-convex nature of the loss function, which causes GD to get trapped in local minima. EM appears more capable of escaping such local optima. In terms of execution times, LIFTCOVER+ is comparable to LIFTCOVER-EM, although it is slower in some cases, particularly when using GD. For some datasets (Bupa, Mondial, Mutagenesis, Pyramidine, Yeast, Triazine, Carcinogenesis), GD is slower by one or more orders of magnitude. It should be noted, however, that the scaling approach applied here provides only a rough approx-

Table 7.3: Average AUC-ROC over the datasets for each configuration: EM with Bayes regularization (EM-Bayes), EM with L1 regularization (EM-L1), EM with L2 regularization (EM-L2), and gradient descent (GD). For each row, the best result is highlighted in bold.

Dataset	EM-Bayes	EM-L1	EM-L2	GD	LIFTCOVER-EM
Financial	0.521	0.459	**0.528**	0.389	0.432
Bupa	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Mondial	0.586	0.547	0.572	0.528	**0.663**
Mutagen.	0.934	0.918	**0.939**	0.594	0.931
Sisyb	**0.500**	**0.500**	**0.500**	**0.500**	**0.500**
Sisya	**0.720**	**0.720**	**0.720**	**0.720**	0.372
Pyrimidine	0.975	0.880	0.910	0.160	**1.000**
Yeast	0.785	0.783	0.785	0.530	**0.786**
Nba	**0.725**	**0.725**	**0.725**	0.675	0.531
Triazine	0.425	0.390	0.430	0.580	**0.713**
UW-CSE	0.976	**0.977**	0.975	0.951	**0.977**
Carcinogen.	0.720	0.692	0.687	0.500	**0.766**
Average	***0.739***	*0.716*	*0.731*	*0.594*	*0.723*

imation, as differences in processor architecture, cache, and pipelining can influence performance.

In conclusion, the experimental analysis shows that incorporating regularization within the EM framework improves model performance, particularly in terms of AUC-PR, without introducing additional computational overhead. L2 and Bayesian regularization proved especially effective, consistently enhancing predictive quality across several datasets. On the other hand, gradient-based optimization did not provide meaningful benefits and often resulted in longer execution times, likely due to the highly non-convex nature of the underlying optimization problem. Regarding the datasets that remain challenging, our results are consistent with previous observations reported for LIFTCOVER [74]. In particular, there does not appear to be a clear correlation between dataset size or number of predicates and system performance: some datasets with a large number of predicates or tuples are handled effectively, while others remain difficult despite having comparable characteristics. This suggests that the observed

Table 7.4: Average AUC-PR over the datasets for each configuration: EM with Bayes regularization (EM-Bayes), EM with L1 regularization (EM-L1), EM with L2 regularization (EM-L2), and gradient descent (GD). For each row, the best result is highlighted in bold.

Dataset	EM-Bayes	EM-L1	EM-L2	GD	LIFTCOVER-EM
Financial	0.155	0.127	**0.169**	0.124	0.126
Bupa	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Mondial	0.717	0.739	0.743	0.712	**0.763**
Mutagen.	0.966	0.964	**0.971**	0.759	**0.971**
Sisyb	**0.286**	**0.286**	**0.286**	**0.286**	**0.286**
Sisya	**0.708**	**0.708**	**0.708**	**0.708**	0.706
Pyrimidine	0.988	0.913	0.947	0.378	**1.000**
Yeast	0.499	0.486	0.497	0.242	**0.502**
Nba	**0.789**	**0.789**	**0.789**	0.743	0.550
Triazine	0.452	0.430	0.463	0.617	**0.734**
UW-CSE	0.341	**0.358**	0.339	0.158	0.220
Carcinogen.	0.687	0.691	**0.722**	0.513	0.672
Average	*0.632*	*0.624*	***0.636***	*0.520*	*0.628*

Table 7.5: Average time in seconds over the datasets for each configuration: EM with Bayes regularization (EM-Bayes), EM with L1 regularization (EM-L1), EM with L2 regularization (EM-L2), and gradient descent (GD). For each row, the best result is highlighted in bold.

Dataset	EM-Bayes	EM-L1	EM-L2	GD	LIFTCOVER-EM
Financial	0.280	0.275	0.278	2.360	**0.235**
Bupa	0.246	0.244	0.244	510.206	**0.243**
Mondial	6.480	**4.841**	5.149	299.875	5.911
Mutagen.	15.073	**10.796**	15.653	266.196	12.770
Sisyb	0.232	0.231	0.233	0.490	**0.226**
Sisya	1.117	1.108	1.111	**0.656**	0.932
Pyrimidine	44.712	**23.408**	25.766	266.310	54.990
Yeast	60.143	52.503	54.288	994.811	**0.502**
Nba	0.658	0.705	0.624	0.913	**0.599**
Triazine	33.648	**23.871**	30.305	276.304	56.690
UW-CSE	74.163	72.961	74.456	75.656	**8.054**
Carcinogen.	14.483	16.826	13.458	778.596	**7.850**
Average	*20.936*	*17.314*	*18.464*	*289.364*	***12.417***

Fig. 7.1: Histograms of average AUC-ROC (a), AUC-PR (b), and time (c) over the datasets for each configuration: EM with Bayes regularization (EM-Bayes), EM with L1 regularization (EM-L1), EM with L2 regularization (EM-L2), and gradient descent (GD). In the time graph, the scale of the X axis is logarithmic.

difficulty may depend on more subtle structural properties of the data, such as the suitability of the adopted language bias for capturing the underlying relational patterns. While LIFTCOVER+ improves performance in several settings, these results indicate that language expressiveness and inductive bias remain critical factors in determining performance across datasets.

7.4 Related work

Lifted inference for PLP under the distribution semantics has been surveyed in [84], where three main approaches are discussed and compared: Lifted Probabilistic Logic Programming (LP^2), lifted inference with aggregation parfactors, and Weighted First Order Model Counting (WFOMC). The authors of [85], instead, focused their survey on lifted graphical models.

Both LIFTCOVER and its extended version LIFTCOVER+ originate from SLIPCOVER [86], an algorithm designed for learning general PLP theories through a search in the clause space, followed by a greedy refinement process that incrementally adds improved clauses to the theory. In contrast to SLIPCOVER, LIFTCOVER, and LIFTCOVER+ simplify the structure search and adopt different methods for parameter learning. While SLIPCOVER relies on EMBLEM [69] to learn the parameters of a probabilistic logic program by applying EM over Binary Decision Diagrams [87], LIFTCOVER and LIFTCOVER+ use special formulas for EM, LBFGS, and gradient descent.

Hierarchical Probabilistic Logic Programs (HPLPs) [76] represents a restricted form of general PLP, where clauses and predicates are organized hierarchically. This structure enables efficient conversion of programs into Arithmetic Circuits (ACs) or deep neural networks, substantially reducing inference costs compared to general PLP. Liftable PLP can be viewed as a further restriction of HPLPs. In this context, LIFTCOVER+ shares conceptual similarities with liftable PLP systems such as PHIL and SLEAHP [76]. PHIL performs parameter learning of HPLPs using gradient descent (DPHIL) or EM (EMPHIL). The program is first converted into a set of parameter-sharing ACs, over which gradient descent or EM is applied in a bottom-up evaluation. On the other hand, SLEAHP learns both the structure and the parameters of HPLPs from data. It starts by generating a large hierarchical logic program from bottom clauses obtained through a language bias [86], and then applies a regularized version

of PHIL to prune irrelevant rules, specifically, those whose parameters are close to 0.

LIFTCOVER+ is also related to PROBFOIL+ [88], an algorithm for parameter and structure learning of ProbLog [55] programs. It employs a hill-climbing search strategy in the space of programs, using a covering loop that iteratively adds new rules to the theory until a global scoring function criterion is met. Each new rule is constructed via a clause search loop that incrementally extends the rule body based on a local heuristic scoring function.

Chapter 8

Probabilistic Answer Set Programming

8.1 Setting

For these works, we adopt the Credal Semantics [89–91], which assigns a meaning to Answer Set Programs extended with probabilistic facts [92] of the form $p :: a$, where p represents the probability associated with the atom a. This notation means that the fact a is included in the program with probability p and excluded with probability $1-p$. Such programs are known as Probabilistic Answer Set Programs (PASP, and we use the same acronym to refer both to the programs themselves and to the programming framework, with the intended meaning clear from the context). Selecting the presence or absence of each probabilistic fact defines a world w whose probability is given by

$$P(w) = \prod_{a \in w} p \cdot \prod_{a \notin w} (1 - p)$$

where $a \in w$ indicates that a is included in w, and $a \notin w$ that it is not. A program with n probabilistic facts has thus 2^n possible worlds, denoted by the set W. Each world corresponds to an Answer Set Program and may yield zero or more answer sets; however, under the CS, it is required that every world has at least one. When this condition holds, the probability of a query q (a conjunction of ground literals) is defined by a lower and an upper bound. A world w contributes to both bounds if every one of its answer sets entails the query (i.e., the query is a cautious consequence). If the

query holds only in some answer sets (a brave consequence), w contributes only to the upper probability. If the query is not satisfied in any answer set, w does not contribute to the probability bounds. In formulas,

$$P(q) = [\underline{\mathrm{P}}(q), \overline{\mathrm{P}}(q)] = [\sum_{w_i \in W | \forall m \in AS(w_i),\ m \models q} P(w_i), \sum_{w_i \in W | \exists m \in AS(w_i),\ m \models q} P(w_i)]. \quad (8.1)$$

The conditional probability of a query q given evidence e, expressed as a conjunction of ground literals, is [93]:

$$\underline{\mathrm{P}}(q \mid e) = \frac{\underline{\mathrm{P}}(q, e)}{\underline{\mathrm{P}}(q, e) + \overline{\mathrm{P}}(not\ q, e)},\ \overline{\mathrm{P}}(q \mid e) = \frac{\overline{\mathrm{P}}(q, e)}{\overline{\mathrm{P}}(q, e) + \underline{\mathrm{P}}(not\ q, e)}. \quad (8.2)$$

For the lower conditional probability $\underline{\mathrm{P}}(q \mid e)$, if $\underline{\mathrm{P}}(q, e) + \overline{\mathrm{P}}(not\ q, e) = 0$ and $\overline{\mathrm{P}}(q, e) > 0$, then $\underline{\mathrm{P}}(q \mid e) = 1$. Similarly, for the upper conditional probability $\overline{\mathrm{P}}(q \mid e)$, if $\overline{\mathrm{P}}(q, e) + \underline{\mathrm{P}}(not\ q, e) = 0$ and $\overline{\mathrm{P}}(not\ q, e) > 0$, then $\overline{\mathrm{P}}(q \mid e) = 0$. Both formulas are undefined if $\overline{\mathrm{P}}(q, e)$ and $\overline{\mathrm{P}}(not\ q, e)$ are 0. To clarify, consider the following example.

Example 8.1.1. The following snippet represents a PASP encoding of a simple graph reachability problem. The first three facts are probabilistic. The rules state that a path exists between X and Y either when they are directly connected or when there exists a node Z such that X is connected to Z and a path exists from Z to Y. The predicates *connected/2* and *nconnected/2* capture the possible presence or absence of a connection between two nodes when an edge is present. Since the program contains three probabilistic facts, there are $2^3 = 8$ possible worlds, listed in Table 8.1. We aim to compute the probability of the query $q = path(1, 4)$. In worlds w_6 and w_7, due to the mutual exclusivity of *connected/2* and *nconnected/2*, multiple answer sets exist. Some of them entail $path(1, 4)$ (when both $connected(1, 2)$ and $connected(2, 4)$ hold), while others do not. Consequently, $path(1, 4)$ is true in at least one but not all answer sets of these worlds, and thus they contribute only to the upper bound. This results in $P(q) = [0, 0.06]$. Observing the evidence $e = edge(2, 4)$ restricts the set of possible worlds to those in which the fact holds. This yields $P(q, e) = [0, 0.06]$, $P(not\ q, e) = [0.24, 0.3]$, hence $\underline{\mathrm{P}}(q \mid e) = 0$ and $\overline{\mathrm{P}}(q \mid e) = 0.2$.

```
0.2::edge(1,2).
0.3::edge(2,4).
0.9::edge(1,3).
```

Table 8.1: Worlds, probabilities, and answer sets of Example 8.1.1. For each world, only literals relevant to the derivation of $path(1,4)$ are shown. The second, third, and fourth columns indicate with 0 or 1 whether each probabilistic fact is false or true in the given world. The LP/UP column specifies whether the world contributes to the lower bound (LP), the upper bound (UP), or does not contribute at all (marked with a dash) to the probability of the query $path(1,4)$.

w_{id}	$edge(1,2)$	$edge(2,4)$	$edge(1,3)$	Answer sets	LP/UP	Probability
w_0	0	0	0	{}	-	0.056
w_1	0	0	1	$\{connected(1,3)\}$, $\{nconnected(1,3)\}$	-	0.504
w_2	0	1	0	$\{connected(2,4), path(2,4)\}$, $\{nconnected(2,4)\}$	-	0.024
w_3	0	1	1	$\{connected(2,4), path(2,4)\}$, $\{nconnected(2,4)\}$	-	0.216
w_4	1	0	0	$\{connected(1,2), path(1,2)\}$, $\{nconnected(1,2)\}$	-	0.014
w_5	1	0	1	$\{nconnected(1,2)\}$, $\{nconnected(1,2)\}$	-	0.126
w_6	1	1	0	$\{connected(1,2), connected(2,4), path(1,4)\}$, $\{connected(1,2), nconnected(2,4)\}$, $\{nconnected(1,2), connected(2,4)\}$, $\{nconnected(1,2), nconnected(2,4)\}$	UP	0.006
w_7	1	1	1	$\{connected(1,2), connected(2,4), path(1,4)\}$, $\{connected(1,2), nconnected(2,4)\}$, $\{nconnected(1,2), connected(2,4)\}$, $\{nconnected(1,2), nconnected(2,4)\}$	UP	0.054

```
path(X,Y):- connected(X,Z), path(Z,Y).
path(X,Y):- connected(X,Y).
connected(X,Y):- edge(X,Y), not nconnected(X,Y).
nconnected(X,Y):- edge(X,Y), not connected(X,Y).
```

8.1.1 Inference in PASP

In PASP, inference can be formulated as a Second Level Algebraic Model Counting Problem (2AMC) [94]. Given a propositional theory T whose variables are divided into two disjoint sets, X_o and X_i, two commutative semirings $R^i = (D^i, \oplus^i, \otimes^i, n_{\oplus^i}, n_{\otimes^i})$ and $R^o = (D^o, \oplus^o, \otimes^o, n_{\oplus^o}, n_{\otimes^o})$, two weight functions, $w_i : lit(X_i) \to D^i$ and w_o :

$lit(X_o) \to D^o$, and a transformation function $f : D^i \to D^o$, 2AMC is defined as:

$$2AMC(T) = \bigoplus\nolimits^{o}_{I_o \in \mu(X_o)} \bigotimes\nolimits^{o}_{a \in I_o} w_o(a) \otimes^o f(\bigoplus\nolimits^{i}_{I_i \in \varphi(T|I_o)} \bigotimes\nolimits^{i}_{b \in I_i} w_i(b))$$

where $\varphi(T \mid I_o)$ denotes the set of assignments to the variables in X_i such that each assignment, combined with I_o, satisfies T, and $\mu(X_o)$ represents all possible assignments to the variables in X_o. 2AMC requires solving two nested Algebraic Model Counting (AMC) [95] tasks: the outer task considers the variables in X_o, and for each assignment to these variables, an inner AMC task is performed over X_i. The two levels are linked by a transformation function f that maps the results of the inner computation to the domain of the outer one. Different instantiations of these components allow 2AMC to represent various reasoning tasks.

To perform inference in PASP, the following configuration was proposed in [96]:

- Inner semiring: $\mathcal{R}^i = (\mathbb{N}^2, +, \cdot, (0,0), (1,1))$ with X_i containing all atoms of the Herbrand base except the probabilistic facts, and w_i mapping *not* q to $(0,1)$ and all other literals to $(1,1)$.

- Outer semiring: $\mathcal{R}^o = ([0,1]^2, +, \cdot, (0,0), (1,1))$, the two-dimensional probability semiring, where X_o contains the atoms corresponding to probabilistic facts, and w_o assigns (p,p) and $(1-p, 1-p)$ to a and *not* a, respectively, for each probabilistic fact $p :: a$, while all other literals are mapped to $(1,1)$.

- Transformation function: $f((n_1, n_2))$ returns the pair (v_{lp}, v_{up}), where $v_{lp} = 1$ if $n_1 = n_2$ and 0 otherwise, and $v_{up} = 1$ if $n_1 > 0$ and 0 otherwise.

2AMC can be efficiently addressed through knowledge compilation [97], a technique commonly employed in probabilistic logic programming. This approach enables a compact representation of the input theory as a tree or graph, allowing probabilistic inference to be performed by traversing the compiled representation.

`aspmc` [98, 99] is one such tool, shown to outperform alternative methods based on projected answer set enumeration [96, 100]. `aspmc` compiles the input theory into a Negation Normal Form (NNF) formula, represented as a tree where internal nodes correspond to conjunctions (AND-nodes) or disjunctions (OR-nodes), and the leaves correspond to the literals of the theory. In particular, `aspmc` targets sd-DNNFs, a class of NNFs that satisfy three additional properties: i) the variables of the children of

AND-nodes are disjoint (decomposability); ii) the conjunction of any pair of children of OR-nodes is logically inconsistent (determinism); and iii) all children of an OR-node depend on the same set of variables (smoothness). In addition, `aspmc` requires the X-firstness property [94], which constraints the order of appearance of variables. Given two disjoint partitions X and Y of the variables in a NNF n, a node is called *pure* if all variables departing from it are members of either X or Y; otherwise, it is called *mixed*. A NNF satisfies the X-firstness property if, for each AND-node, either all its children are pure, or at most one child is mixed while the others are pure nodes whose variables belong to X.

We explored two different approaches for performing parameter learning in PASP, each of which is described in the following sections.

8.2 Symbolic Parameter Learning in PASP

We adopt the Learning from Interpretations framework proposed in [101], which we briefly recall here for clarity. A PASP is denoted by $\mathcal{P}(\Pi)$, where Π represents the set of parameters to be learned. These parameters correspond to the probabilities associated with a subset of the probabilistic facts, which we refer to as *learnable facts*. Some probabilistic facts may have fixed probabilities, i.e., not all facts are subject to learning.

A partial interpretation $I = \langle I^+, I^- \rangle$ consists of two sets I^+ and I^-, representing atoms that are true and false, respectively. It is called partial because it may specify the truth value of only some atoms. Given a partial interpretation I, we define the interpretation query as $q_I = \bigwedge_{i^+ \in I^+} i^+ \bigwedge_{i^- \in I^-} not\ i^-$. The probability of an interpretation I, denoted $P(I)$, is the probability of its interpretation query. Since the program is interpreted under the CS, this probability is expressed as a range. Given a PASP $\mathcal{P}(\Pi)$, the lower and upper probability for a partial interpretation I are defined as

$$\overline{\mathrm{P}}(I \mid \mathcal{P}(\Pi)) = \sum_{w \in \mathcal{P}(\Pi) \mid \exists m \in AS(w),\ m \models I} P(w),$$

$$\underline{\mathrm{P}}(I \mid \mathcal{P}(\Pi)) = \sum_{w \in \mathcal{P}(\Pi) \mid \forall m \in AS(w),\ m \models I} P(w).$$

Formally, given a PASP $\mathcal{P}(\Pi)$ and a set of (partial) interpretations $\mathcal{I}$, the goal of the

parameter learning task is to identify a probability assignment to the probabilistic facts that maximizes the likelihood, i.e., the product of the lower (or upper) probabilities of the partial interpretations:

$$\Pi^* = \arg\max_{\Pi} \underline{\mathrm{P}}(\mathcal{I} \mid \mathcal{P}(\Pi)) = \arg\max_{\Pi} \prod_{I \in \mathcal{I}} \underline{\mathrm{P}}(I \mid \mathcal{P}(\Pi))$$

Alternatively, this optimization considers the log-likelihood, which is often preferable to avoid numerical instability when dealing with products of many small probabilities. This leads to the equivalent formulation:

$$\Pi^* = \arg\max_{\Pi} \log(\underline{\mathrm{P}}(\mathcal{I} \mid \mathcal{P}(\Pi))) = \arg\max_{\Pi} \sum_{I \in \mathcal{I}} \log(\underline{\mathrm{P}}(I \mid \mathcal{P}(\Pi))) \tag{8.3}$$

The maximum achievable log-likelihood value is zero, which occurs when every interpretation has a probability of 1.

Because PASP defines probabilities in terms of lower and upper bounds, one must choose whether to optimize the lower or upper probability. Importantly, these two objectives are not always aligned: maximizing one does not necessarily maximize the other. For example, consider the following program: $\{\{q : -a, b, not\ nq\}, \{nq : -a, b, not\ q\}\}$ where a and b are probabilistic facts with probabilities p_a and p_b, respectively. Suppose the interpretation is $I = \langle\{q\}, \{\}\rangle$. In this case, $\underline{\mathrm{P}}(I) = 0$ whereas $\overline{\mathrm{P}}(I) = p_a \cdot p_b$. As a result, any choice of p_a and p_b yields the same lower probability (zero), while only the assignment $p_a = 1$ and $p_b = 1$ will maximize $\overline{\mathrm{P}}(I)$. This illustrates the asymmetry between optimizing over lower and upper bounds in PASP learning.

Example 8.2.1. Consider the program shown in Example 8.1.1. Assume we have two interpretations: $I_0 = \langle\{path(1,3)\}, \{path(1,4)\}\rangle$ and $I_1 = \langle\{path(1,4)\}, \{\}\rangle$. These correspond to two interpretation queries: $q_{I_0} = path(1,3),\ not\ path(1,4)$ and $q_{I_1} = path(1,4)$. Let Π denote the set of parameters associated with the probabilistic facts involved. The parameter learning task reduces to the following optimization problem: $\Pi^* = \arg\max_{\Pi}(\log(P(q_{I_0} \mid \Pi)) + \log(P(q_{I_1} \mid \Pi)))$.

Authors of [101] addressed the task of learning the probabilities of ground probabilistic facts using model enumeration instead of knowledge compilation, and introduced an approach based on the Expectation-Maximization algorithm to solve it. When

the objective is to optimize the upper probability, the method proceeds as follows (the case for the lower probability is handled analogously, with only the bound differing). In the expectation step, for each probabilistic fact a_i whose probability is to be learned, the expected contributions of the fact being false and true are computed as:

$$\overline{\mathrm{E}}[a_{i0}] = \sum_{I \in \mathcal{I}} \overline{\mathrm{P}}(not\ a_i \mid I), \quad \overline{\mathrm{E}}[a_{i1}] = \sum_{I \in \mathcal{I}} \overline{\mathrm{P}}(a_i \mid I). \tag{8.4}$$

These expectations are used in the maximization step to update each parameter Π_i as:

$$\Pi_i = \frac{\overline{\mathrm{E}}[a_{i1}]}{\overline{\mathrm{E}}[a_{i0}] + \overline{\mathrm{E}}[a_{i1}]} = \frac{\sum_{I \in \mathcal{I}} \overline{\mathrm{P}}(a_i \mid I)}{\sum_{I \in \mathcal{I}} \overline{\mathrm{P}}(not\ a_i \mid I) + \overline{\mathrm{P}}(a_i \mid I)}. \tag{8.5}$$

We introduce two algorithms for the parameter learning task. Both approaches rely on extracting symbolic equations from the NNF [97] representing a query. Since both algorithms share this symbolic extraction step, we describe it first. We focus on computing the upper probability, while the treatment of the lower bound is analogous.

8.2.1 Extracting Equations from a NNF

The upper probability of a query q can be expressed as a sum of products, as shown in Equation 8.1. By treating the probabilities associated with facts as symbolic variables instead of numeric values, we obtain a nonlinear symbolic expression for the query's probability in which each variable corresponds to the parameter of a learnable fact. We denote this expression by $f_{up}(\Pi)$, where Π is the set of parameters. Its general form is: $f_{up}(\Pi) = \sum_{w_i} \prod_{a_j \in w_i} p_j \prod_{a_j \notin w_i} (1 - p_j) \cdot k_i$. Here k_i is the contribution of the probabilistic facts with fixed probability in world w_i. We formulate this extraction as a 2AMC problem, relying on the inner semiring and transformation function described in [96] and in the previous section. The inner semiring yields two values: one for the lower bound and one for the upper bound. To extract the equation, we extend the sensitivity semiring from [95] to handle a pair of values: $R^o = (\mathbb{R}[X], +, -, (0, 0), (1, 1))$

with

$$w_o(l) = \begin{cases} (p, p) & \text{for a p.f. } p :: a \text{ with fixed probability and } l = a \\ (1-p, 1-p) & \text{for a p.f. } p :: a \text{ with fixed probability and } l = \mathit{not}\ a \\ (\pi_a, \pi_a) & \text{for a learnable fact } \pi_a :: a \text{ and } l = a \\ (1-\pi_a, 1-\pi_a) & \text{for a learnable fact } \pi_a :: a \text{ and } l = \mathit{not}\ a \\ (1, 1) & \text{otherwise} \end{cases}$$

Here, p.f. denotes a probabilistic fact, $\mathbb{R}[X]$ is the set of real-valued functions parameterized by X, and π_a is the symbolic probability of the learnable fact a. Evaluating the symbolic expression with specific parameter values results in the actual query probability for the corresponding setting of the learnable facts.

Example 8.2.2. Considering Example 8.1.1, with query $path(1, 4)$ and associating a variable π_{xy} with each $edge(x, y)$ probabilistic fact (and considering them as learnable), the symbolic equation for its upper probability is $\pi_{12} \cdot \pi_{24} \cdot \pi_{13} + \pi_{12} \cdot \pi_{24} \cdot (1 - \pi_{13})$, that can be simplified to $\pi_{12} \cdot \pi_{24}$.

8.2.2 Solving Parameter Learning with Constrained Optimization

The symbolic equations extracted from the NNF can be used to solve the parameter learning task. The objective is to maximize the sum of the log-probabilities of the interpretations (see Equation 8.3), where the tunable parameters are those associated to the facts whose probability should be learned. The main steps are outlined in Algorithm 2. For each interpretation, we use function GetEquationFromNNF to extract a symbolic expression representing its probability from the NNF structure, treating each learnable fact's probability as a variable. To reduce the size of the equation and thus the computational complexity, the resulting equation is simplified using the Simplify function. Next, we formulate a constrained nonlinear optimization problem, where the objective function is the sum of the equations representing the log probabilities of the interpretations. This step is performed with function SolveOptimizationProblem. Constraints are added to ensure that all parameters lie within the valid probability range, that is, between 0 and 1. This formulation allows us to leverage off-the-shelf

Algorithm 2 Function LearningOPT: solving the parameter learning task targeting the upper probability with constrained optimization in a PASP P with learnable probabilistic facts Π and interpretations I.

```
1: function LearningOPT(P, I)
2:     eqsList ← ∅
3:     for i ∈ I do
4:         eq ← GetEquationFromNNF(i)
5:         eqsList ← eqsList ∪ Simplify(eq)
6:     end for
7:     Π* ← SolveOptimizationProblem(eqsList)
8:     return Π*
9: end function
```

optimization solvers, avoiding the need to implement a custom solution.

Solving Parameter Learning with Expectation Maximization

As an alternative approach, we introduce an algorithm based on EM, following the formulation in Equation 8.4 and Equation 8.5. The procedure is outlined in Algorithm 3. For each interpretation I, we add its interpretation query q_I into the program. To compute $\overline{\mathrm{P}}\,(a_j \mid I_k)$ for each learnable probabilistic fact a_j and each interpretation I_k we first compute $\overline{\mathrm{P}}\,(a_j, I_k)$ and $\underline{\mathrm{P}}\,(not\ a_j, I_k)$. These values are then combined using Equation 8.2 to derive $\overline{\mathrm{P}}(a_j \mid I_k)$, and analogously for the lower bound $\underline{\mathrm{P}}\,(a_j \mid I_k)$. For each probabilistic query, we extract the corresponding symbolic equation with function GetEquationsFromNNF. These expressions are evaluated iteratively until convergence of the EM procedure. Lines 5–10 of the algorithm define the main loop, alternating between the expectation phase (Expectation), the maximization phase (Maximization), and the computation of the current log-likelihood (ComputeLL). By default, convergence is determined when the change in log-likelihood between two consecutive iterations falls below $5 \cdot 10^{-4}$, though this threshold is configurable by the user. Let n_p be the number of learnable probabilistic facts and n_i the number of interpretations. We need to extract equations for $2 \cdot n_p \cdot n_i$ distinct queries. However, this overhead is mitigated by the fact that the structure of the program remains unchanged across iterations, as only the probabilistic parameters are updated. Therefore, the NNF representation needs to be constructed only once. The operations involved in query evaluation can be cached and reused in subsequent iterations, avoiding the need to rebuild the NNF at every step.

Example 8.2.3. Consider Example 8.2.1 and its two interpretation queries, q_{I_0} and

Algorithm 3 Function LEARNINGEM: solving the parameter learning task targeting the upper probability with Expectation Maximization in a PASP P with learnable probabilistic facts Π and with interpretations I.

```
1: function LearningEM(P, I)
2:     eqsList ← GetEquationsFromNNF(π, I)
3:     ll_0 ← 0
4:     ll_1 ← 1
5:     while (ll_1 − ll_0) > 5 · 10^−4 do                ▷ Loop until convergence.
6:         ll_0 ← ll_1
7:         E ← Expectation(eqsList)
8:         Π ← Maximization(E)
9:         ll_1 ← ComputeLL(eqsList, Π)
10:    end while
11:    return Π
12: end function
```

q_{I_1}. If we consider their symbolic equations we have $\pi_{12} \cdot \pi_{24}$ (see Example 8.2.2) and $\pi_{13} \cdot (\pi_{12} + \pi_{24} - \pi_{12} \cdot \pi_{24})$, where with π_{xy} we indicate the probability of $edge(x, y)$. If we compactly denote with Π the set of all the probabilities (i.e., all the π_{xy}), the optimization problem of Equation 8.3 requires solving $\Pi^* = \arg\max_{\Pi}(\log(\pi_{12} \cdot \pi_{24}) + \log(\pi_{13} \cdot (\pi_{12} + \pi_{24} - \pi_{12} \cdot \pi_{24})))$. The optimal solution is to set the values of all the π_{xy} to 1, obtaining a log-likelihood of 0. Note that, in general, it is not always possible to obtain a LL of 0.

8.2.3 Experiments

We conducted our experiments on a machine equipped with 16GB of RAM and a 2.40GHz processor, with an 8-hour time limit per run. The experiments were designed to address multiple goals: i) identifying which algorithm performs faster; ii) assessing which method more effectively solves the optimization problem defined in Equation 8.3 (recalling that the maximum of the log-probability sum is 0, which occurs when all partial interpretations are assigned probability 1); iii) examining whether different initializations of the learnable probabilities influence execution time; and iv) comparing our methods with PASTA, the algorithm based on projected answer set enumeration introduced in [101]. We focused on PASTA as it is currently the only available approach for parameter learning in PASP under the CS.

The system relies on several open-source components: `aspmc` [98] is used as backend solver for the computation of the formula, SciPy version 1.13.0 is the optimization solver [102], and SymPy [103] version 1.12 is employed to simplify equations. The

system is thus fully open source[1].

For the approach based on constrained optimization (Section 8.2.2), we tested two nonlinear optimization algorithms available in SciPy: COBYLA [104] and SLSQP [105]. COBYLA (Constrained Optimization BY Linear Approximations) is a derivative-free method that approximates the objective and constraint functions linearly at each iteration. In contrast, SLSQP (Sequential Least Squares Programming) solves a sequence of quadratic approximations of the original problem by minimizing a least-squares objective subject to constraints.

The algorithm based on Expectation Maximization, introduced in Section 8.2.2, is referred to as EM in the experimental results.

We considered four datasets in our experimental evaluation.

The *coloring* dataset models the graph coloring problem. In this setting, each node is assigned a color, and a configuration is valid only if two adjacent nodes do not share the same color. Rather than discarding invalid solutions outright, we retain them and explicitly label them as invalid. This characteristic makes the dataset particularly useful for evaluating our approach in scenarios where the number of answer sets per possible world increases significantly. All instances in this dataset are constructed using the following rules:

```
red(X) :- node(X), not green(X), not blue(X).
green(X) :- node(X), not red(X), not blue(X).
blue(X) :- node(X), not red(X), not green(X).
e(X,Y) :- edge(X,Y).
e(Y,X) :- edge(Y,X).
c0 :- e(X,Y), red(X), red(Y).
c1 :- e(X,Y), green(X), green(Y).
c2 :- e(X,Y), blue(X), blue(Y).
valid :- not c0, not c1, not c2.
```

The objective in this dataset is to learn the probabilities associated with the *edge*/2 facts, given an increasing number of interpretations. Each interpretation specifies the observed colors assigned to certain edges and indicates whether the corresponding configuration is *valid* or not. We considered two variants of complete graphs, one with

[1]Datasets and source code available at https://github.com/damianoazzolini/aspmc and on Zenodo with DOI 10.5281/zenodo.12667046.

4 nodes (*coloring4*) and one with 5 (*coloring5*). For each variant, interpretations were generated containing between 3 and 4 atoms.

The *path* dataset models a path reachability problem, where each instance is based on the rules presented in Example 8.1.1. We included this dataset because the underlying graph structure can be used to represent a wide range of problems. The learning task consists of inferring the probabilities of the *edge*/2 facts, given a set of observations specifying whether certain paths are reachable. We considered randomly generated connected graphs with 10 (*path10*) and 15 (*path15*) edges. In this case, the interpretations contain between 1 and 3 atoms.

The *shop* dataset simulates the shopping behavior of multiple individuals, where a selection of products is available and certain combinations are mutually exclusive (i.e., some products cannot be purchased together). Each instance represents a shopping scenario involving a set of people and the products they buy. Below is an example of an instance with two people:

```
bought(spaghetti,john):-shops(john), not bought(steak,john).
bought(steak,john):-shops(john), not bought(spaghetti,john).
bought(spaghetti,mary):-shops(mary), not bought(beans,mary).
bought(beans,mary):-shops(mary), not bought(spaghetti,mary).
bought(spaghetti):- bought(spaghetti,_).
bought(steak):- bought(steak,_).
:- bought(spaghetti), bought(steak).
```

The goal is to learn the probabilities of the *shops*/1 learnable facts, based on an increasing number of both observed purchases (and non-purchases) and number of people. We considered instances with 4 (*shop4*), 8 (*shop8*), 10 (*shop10*), and 12 (*shop12*) individuals. The interpretations associated with each instance have a random length between 1 and 10 atoms. This dataset is particularly useful to evaluate the behavior of the learning algorithms in the presence of constraints, which prune part of the solution space by eliminating incompatible combinations of facts.

The *smoke* dataset, adapted from [106], represents a network in which individuals may smoke, and their smoking behavior can be influenced by their peers. This dataset is a standard benchmark in the field of probabilistic logic programming. Each instance defines a set of people and influences relationships among them. An example with two individuals is shown below:

```
0.1::asthma_f(1). 0.4::asthma_fact(1). 0.3::stress(1).
0.1::asthma_f(2). 0.4::asthma_fact(2). 0.3::stress(2).
```

```
0.2 :: predisposition.

smokes(X) :- stress(X).
smokes(X) :- influences(Y, X), smokes(Y).
asthma_rule(X):- smokes(X), asthma_fact(X).

asthma(X) :- asthma_f(X).
asthma(X) :- asthma_rule(X).

ill(X)  :-smokes(X), asthma(X), not n_ill(X).
n_ill(X):-smokes(X), asthma(X), predisposition, not ill(X).
```

The learnable facts in this setting are *influences*/2, and the goal is to infer their probabilities given observations about which individuals are ill, i.e., *ill*/1 facts are observed. We considered instances with 3 (*smoke2*), 4 (*smoke4*), and 6 (*smoke6*) people. Each interpretation has a randomly chosen size between 1 and 3 atoms.

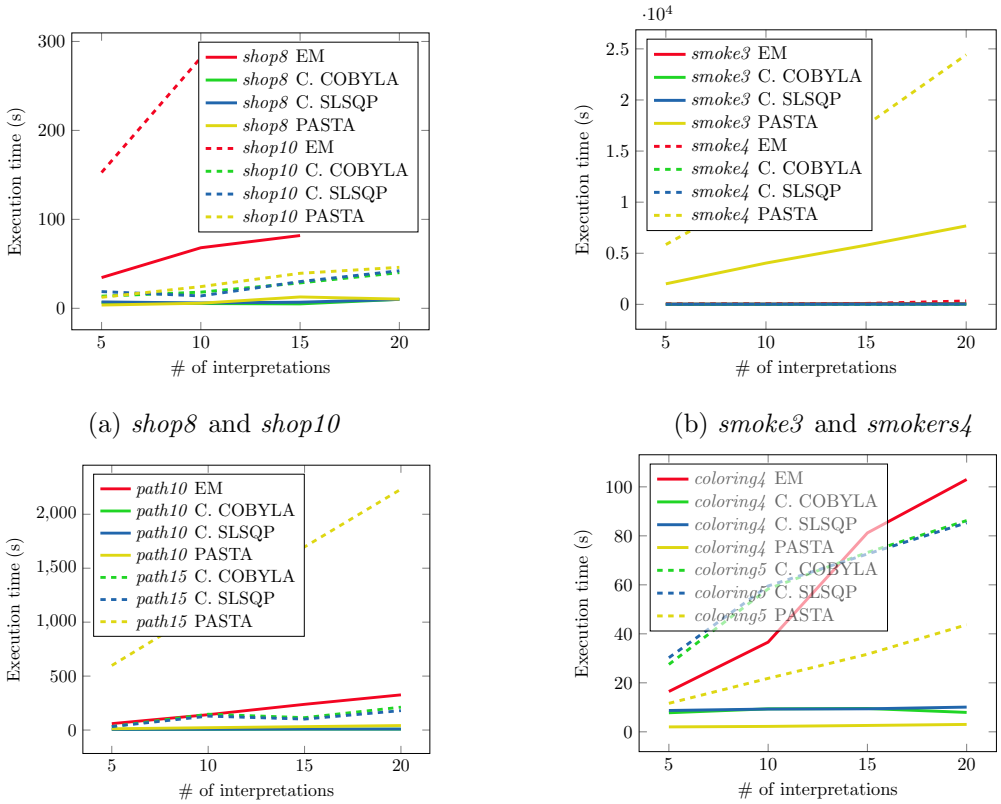
Unless otherwise stated, the initial probability for each learnable fact is set to 0.5. For every instance, we generated configurations containing 5, 10, 15, and 20 interpretations. The atoms included in each interpretation are sampled uniformly at random from the Herbrand base of the corresponding program, with a subset of them randomly selected to appear as negated literals.

We first compare the execution times of the four proposed algorithms. The results are shown in Figure 8.1. Among the tested methods, COBYLA and SLSQP exhibit comparable and consistently low execution times. In contrast, EM and PASTA are generally slower, with PASTA being faster only in *coloring4* and *coloring5*. Due to time and memory constraints, some algorithms fail to solve specific configurations. In particular, EM exceeds memory limits and fails on all configurations of *shop12*, on *shop10* with 15 and 20 interpretations, on *shop8* with 20 interpretations, on all instances of *coloring5*, and on *path15*. PASTA, on the other hand, fails to solve *smoke6* with any number of interpretations and *smoke5* with 10, 15, and 20 interpretations, due to exceeding the time limit. In contrast, both constrained optimization algorithms (COBYLA and SLSQP) successfully solve all instances within the allocated time limit.

In addition to runtime, we also evaluated the algorithms based on the final log-likelihood. Table 8.2 reports the LL values for selected instances: *coloring4*, *path10*,

Table 8.2: Final log-likelihood (LL) values for the tested algorithms on six selected instances with the initial probability of the learnable facts set to 0.5. The column # int. contains the number of interpretations considered, the column EM contains the results obtained with Expectation Maximization, columns C. COBYLA and C. SLSQP stands for constrained optimization solved with, respectively, COBYLA and SLSQP, and the column PASTA contains the results obtained with the PASTA solver.

	coloring4				*shop8*			
# int.	EM	C. COBYLA	C. SLSQP	PASTA	EM	C. COBYLA	C. SLSQP	PASTA
5	-0.016	0.000	0.000	-34.539	-1.389	-12.710	-1.385	-34.077
10	-0.049	0.000	0.000	-55.262	-2.257	0.000	0.000	-63.268
15	-0.044	0.000	0.000	-96.709	-1.923	0.000	-1.902	-75.986
20	-0.030	-36.775	0.000	-124.340	-1.388	-72.524	-0.011	-127.679
	path10				*smoke4*			
# int.	EM	C. COBYLA	C. SLSQP	PASTA	EM	C. COBYLA	C. SLSQP	PASTA
5	-0.002	0.000	0.000	-27.631	-9.430	-9.423	-9.423	-20.723
10	-0.004	0.000	0.000	-48.354	-25.164	-25.160	-25.160	-48.354
15	-0.007	0.000	0.000	-75.985	-50.023	-50.020	-50.020	-96.709
20	-0.004	0.000	0.000	-103.616	-64.576	-64.571	-64.571	-110.524
	smoke3				*shop4*			
# int.	EM	C. COBYLA	C. SLSQP	PASTA	EM	C. COBYLA	C. SLSQP	PASTA
5	-14.633	-14.630	-14.630	-27.631	-1.462	-150.606	0.000	-20.633
10	-36.711	-36.708	-36.708	-69.078	-5.669	0.000	0.000	-54.992
15	-35.172	-35.166	-35.166	-69.078	-7.283	-49.346	0.000	-74.710
20	-72.872	-72.869	-72.869	-110.524	-2.770	0.000	-2.768	-101.824



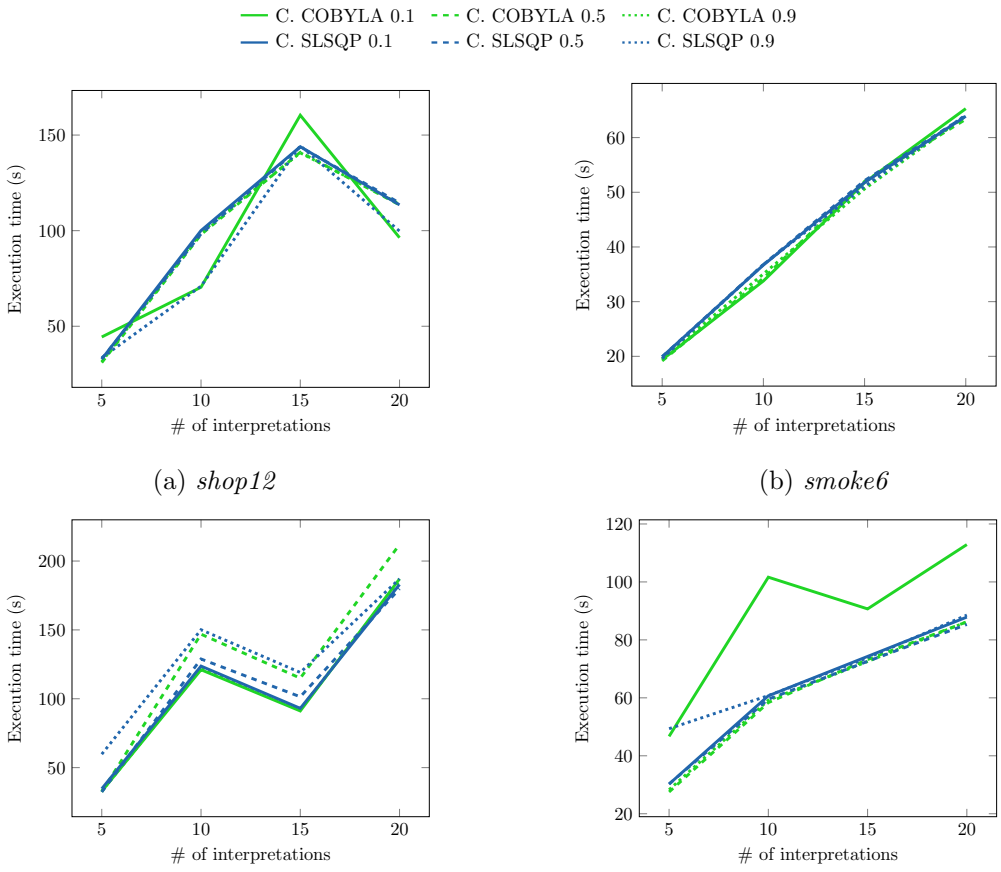
(a) *shop8* and *shop10*

(b) *smoke3* and *smokers4*

(c) *path10* and *path15*

(d) *coloring4* and *coloring5*

Fig. 8.1: Execution times for EM, constrained optimization solved with COBYLA and SLSQP, and PASTA, by increasing the number of interpretations. For *path15* and *coloring5* the line for EM is missing since it cannot solve any instance. The initial probabilities for learnable facts are set to 0.5.



(a) *shop12* (b) *smoke6*

(c) *paths15* (d) *coloring5*

Fig. 8.2: Execution times of constrained optimization solved with COBYLA and SLSQP on *coloring5*, *path15*, *shop12*, and *smoke6* with 0.1, 0.5, and 0.9 as initial values for the learnable facts.

shop4, *shop8*, *smoke3*, and *smoke4*. SLSQP consistently achieves the highest LL across multiple datasets, specifically on *coloring4*, *shop4*, and *shop8*, while COBYLA often reaches suboptimal solutions. EM performs comparably to SLSQP in some cases. PASTA shows the weakest performance, failing to reach a log-likelihood of 0 (the theoretical optimum), while the other approaches succeed, particularly those based on constrained optimization.

We further investigated the impact of different initial probability values on execution time for the constrained optimization algorithms. Results are shown in Figure 8.2. For SLSQP, the initial values have negligible impact on performance. However, COBYLA appears more sensitive to initialization: for example, for *coloring5* with 10 interpretations, setting the initial values to 0.1 results in a runtime of 100 seconds, while setting them to 0.5 or 0.9 reduces it to approximately 60 seconds. We extended this evaluation to EM, but found that the initial probability values do not influence its execution time. PASTA, in its current implementation, does not allow setting initial probabilities different from 0.5.

In summary, the constrained optimization approach demonstrates superior performance both in terms of efficiency and final log-likelihood, outperforming both EM and PASTA. While EM achieves competitive LL values, it suffers from high memory consumption, which limits its applicability on larger instances. PASTA, although more memory-efficient, is both slower and less accurate than the other approaches. The experimental evaluation focuses on relatively small instances, reflecting the intrinsic computational characteristics of PASP under the credal semantics. In practice, the computational cost depends on grounding and answer set enumeration, and grows with the number of learnable facts, the size of the Herbrand base, and the number of partial interpretations. With contemporary technology, instances involving a limited number of probabilistic facts and interpretations can be handled, while scaling to substantially larger domains remains challenging without further structural restrictions or approximations. While decision trees are more scalable, they lack the ability to represent recursive rules, constraints, and non-monotonic reasoning patterns that are naturally captured in PASP. As a result, our approach is better suited for domains where symbolic structure and logical dependencies are central, rather than for large-scale purely statistical learning tasks.

8.3 The Gradient Semiring for PASP and Its Application to Parameter Learning

Semirings have recently emerged as a general framework for a variety of reasoning and learning tasks [94, 95]. In this context, we propose an approach that extends the gradient semiring to address the parameter learning task in PASP.

8.3.1 The Gradient Semiring

We refer to the probabilistic facts whose probabilities should be learned as *tunable*, in contrast to those with fixed probabilities, which we refer to as *fixed*.

In the context of PLP, the gradient semiring is defined as [95] $\mathcal{R}_g = ([0,1] \times \mathbb{R}^n, \oplus, \otimes, (0,\overline{0}), (1,\overline{1}))$, where each element is a pair consisting of the probability value and its gradient w.r.t. the parameters, respectively, $(a, e_a) \oplus (b, e_b) = (a + b, e_a + e_b)$ and $(a, e_a) \otimes (b, e_b) = (a \cdot b, a \cdot e_b + e_a \cdot b)$. The weight function is:

$$w(l) = \begin{cases} (\pi_i, e_i) & \text{if } l = a_i \text{ with } \pi_i :: a_i \text{ tunable} \\ (1 - \pi_i, -e_i) & \text{if } l = \mathit{not}\ a \text{ with } \pi_i :: a_i \text{ tunable} \\ (\pi_i, \overline{0}) & \text{if } l = a_i \text{ with } \pi :: a_i \text{ fixed} \\ (1 - \pi_i, \overline{0}) & \text{if } l = \mathit{not}\ a_i \text{ with } \pi_i :: a_i \text{ fixed} \end{cases}$$

where e_i is a one-hot vector containing 1 at position i and 0 elsewhere.

To extend this construction to the PASP setting, we define an outer semiring as a 4-tuple. The first two elements are the lower and upper probability, while the remaining two are arrays of length n, where n is the number of tunable probabilistic facts, that store at position i the derivative of the query w.r.t. the parameter of probabilistic fact a_i. This results in a 2AMC task characterized by the following components:

- Inner semiring: $\mathcal{R}_i = (\mathbb{N}^2, +, \cdot, (0,0), (1,1))$, i.e., the inner semiring for inference in PASP.
- Outer semiring: $\mathcal{R}_o = ([0,1]^2 \times \mathbb{R}^2, \oplus, \otimes, (0,0,\overline{0},\overline{0}), (1,1,\overline{0},\overline{0}))$
  where $(l_a, u_a, e_a^l, e_a^u) \oplus (l_b, u_b, e_b^l, e_b^u) = (l_a + l_b, u_a + u_b, e_a^l + e_b^l, e_a^u + e_b^u)$,
  $(l_a, u_a, e_a^l, e_a^u) \otimes (l_b, u_b, e_b^l, e_b^u) = (l_a \cdot l_b, u_a \cdot u_b, l_a \cdot e_b^l + l_b \cdot e_a^l, u_a \cdot e_b^u + u_b \cdot e_a^u)$,
  i.e., the doubled gradient semiring for PLP.

- Transformation function:

$$f((v_l, v_u)) = \begin{cases} (1, 1, \overline{0}, \overline{0}) & \text{if } v_l = v_u \text{ and } v_u > 0 \\ (0, 1, \overline{0}, \overline{0}) & \text{if } v_u > v_l \\ (0, 0, \overline{0}, \overline{0}) & \text{otherwise} \end{cases}$$

- Inner weight function:

$$w_i(l) = \begin{cases} (0, 1) & \text{if } l = \textit{not } q \\ (1, 1) & \text{otherwise} \end{cases}$$

- Outer weight function:

$$w_o(l) = \begin{cases} (\pi_i, \pi_i, e_i, e_i) & \text{if } l = a \text{ with } \pi_i :: a_i \text{ tunable} \\ (1 - \pi_i, 1 - \pi_i, -e_i, -e_i) & \text{if } l = \textit{not } a \text{ with } \pi_i :: a_i \text{ tunable} \\ (\pi_i, \pi_i, \overline{0}, \overline{0}) & \text{if } l = a \text{ with } \pi_i :: a_i \text{ fixed} \\ (1 - \pi_i, 1 - \pi_i, \overline{0}, \overline{0}) & \text{if } l = \textit{not } a \text{ with } \pi_i :: a_i \text{ fixed} \end{cases}$$

8.3.2 Parameter Learning

For this work, we extend the parameter learning setting of EMBLEM [107] to PASP. We focus on the optimization of the upper probability; the formulation for the lower probability is analogous. To estimate the parameters, we adopt a Mean Squared Error (MSE) minimization approach, aiming to find the probability values for the probabilistic facts that best match the observed data.

Let P be a PASP containing probabilistic facts with unknown probabilities, and Π the set of such probabilities. Given two sets of ground atoms, E^+ *positive examples* and E^- *negative examples*, the learning objective is to find an assignment Π^* for Π that minimizes the MSE:

$$\arg\min_{\Pi} MSE = \arg\min_{\Pi} \frac{1}{|E|}\Big(\sum_{e^+ \in E^+} (\overline{\mathrm{P}}(e^+) - 1)^2 + \sum_{e^- \in E^-} (\overline{\mathrm{P}}(e^-))^2 \Big). \tag{8.6}$$

This is an alternative definition to the ones for the same task presented in [108–110]

and in the previous section. Here, the probability of positive (negative) examples $\overline{\mathrm{P}}(e^+)$ ($\overline{\mathrm{P}}(e^-)$) is defined as the probability of the query given a mega-example, i.e., a set of ground atoms.

Unlike EMBLEM, which is restricted to dynamically stratified programs where every possible world yields a unique stable model (coinciding with the Well-Founded Model [45]), our method generalizes to handle also non-stratified programs, where a world may admit multiple stable models.

We use $\#positive(I)$ and $\#negative(I)$ to indicate whether a mega-example is negative or positive, respectively, where I is the identifier of the example. Individual atoms of the mega-example are specified via facts of the form $\#atom(I, atom)$. To clarify, consider Example 8.3.1.

Example 8.3.1. The *Bongard* dataset [64][2], derives from one of Bongard's visual concept learning problems and features a set of pictures composed of geometric shapes, such as triangles, squares, and circles, some of which are inside others. Each picture is described by a set of logical facts. For example, a picture with a triangle $o1$ inside a circle $o2$ can be described by the following facts: $triangle(o1)$, $circle(o2)$, $inside(o1, o2)$. Here, a mega-example describes a picture with different geometric shape configurations. A negative and a positive mega-example are shown in the snippet below.

```
#negative(1).
#atom(1,square(o2)).
#atom(1,square(o1)).
#atom(1,inside(o1,o2)).

#positive(2).
#atom(2,circle(o2)).
#atom(2,triangle(o1)).
#atom(2,config(o1,up)).
#atom(2,inside(o1,o2)).
```

[2]Dataset available at: https://cplint.eu/example/learning/bongard.pl

8.3.3 Algorithm

Focusing on the upper probability, we try to minimize the MSE of the examples via gradient descent. Let π_j denote the parameter (i.e., probability) associated to probabilistic fact a_j. The gradient of the MSE w.r.t. π_j is given by [111]:

$$\frac{\partial MSE}{\partial \pi_j} = \frac{2}{|E|} \sum_{e \in E} (\overline{\mathrm{P}}(e) - P_T(e)) \cdot \frac{\partial \overline{\mathrm{P}}(e)}{\partial \pi_j} \tag{8.7}$$

The probability and its gradient can be efficiently computed using the gradient semiring. The parameters are then updated iteratively, following the gradient descent procedure, until a convergence criterion is met. The entire procedure is outlined in Algorithm 4. The algorithm begins with function InitProbabilites, which initializes the values of the probabilistic facts. At each iteration, for every example, we compute both the probability and its gradient using the gradient semiring (function GradientSemiring), and store the result in the losses list. Once all examples have been processed, the parameters are updated according to Equation 8.7, and convergence is assessed with function CheckConvergence.

We implemented Algorithm 4 on top of the `aspmc` solver [112], and refer to our implementation as **PASP-GD**. In function GradientSemiring, each PASP corresponding to an example is transformed into an sd-DNNF [113]. An important aspect is that the compilation step is only performed once, during the first iteration. Since the structure of the circuit depends solely on the logical structure of the program (i.e., its rules), and not on the actual probability values, it remains fixed throughout the learning process. In subsequent iterations, the circuit is reused: only the probability values are updated, and the circuit is simply re-evaluated with the new parameters.

8.3.4 Experiments

We carried out an experimental evaluation to assess the learning performance of the proposed algorithm. We employed 5-fold cross-validation to ensure a reliable evaluation and mitigate variance. The learning procedure terminates either when a convergence criterion is met, defined as a minimum improvement in log-likelihood below a fixed tolerance, or when a maximum number of iterations is reached. All experiments were conducted on a machine equipped with an AMD EPYC 9454 48-core CPU. The memory

Algorithm 4 PASP-GD. Parameter Learning on a PASP P with examples $E = (E^+, E^-)$, learning rate η, and convergence tolerance ϵ.

function ParameterLearning(P,E,ϵ)
  $converged \leftarrow false$
  $losses \leftarrow [\,]$
  $\Pi \leftarrow$ InitProbabilities
  $ll_0 \leftarrow -\infty$
  while not converged **do**
    for $e \in E$ **do**
      $(lp, up, \underline{e}, \overline{e}) \leftarrow$ GradientSemiring(e,P,Π)
      if target = lower **then**
        $p = lp$, $e = \underline{e}$
      else
        $p = up$, $e = \overline{e}$
      end if
      $losses \leftarrow losses \cup (p - P_T(e)) \cdot e$
    end for
    $\Pi^o = \Pi - 2 \cdot \eta \cdot \mu(losses)$ ▷ μ computes the mean
    $\Pi \leftarrow$ Clip(Π^o) ▷ Clip to $[0, 1]$.
    $ll_1 \leftarrow$ ComputeLL(P,Π)
    $converged \leftarrow$ CheckConvergence(ll_0,ll_1,ϵ)
    $ll_0 \leftarrow ll_1$
  end while
end function

limit was set to 32 GB of RAM, and the time limit was set to one hour per run.

Datasets

We tested PASP-GD on two different datasets: one based on a non-stratified program and the other on a stratified one. For clarity, we briefly recall the distinction between stratified and non-stratified programs, which is based on the definition of dependency graph. The dependency graph D_P of a program P is a directed graph [114] where each node corresponds to a ground atom in P. For each rule in the grounding of P, with head atom p and body containing an atom q, the graph includes a directed edge from p to q, labeled as positive if q appears in a positive literal, or negative if it appears negated. A program is said to be *non-stratified* if its dependency graph contains a cycle involving at least one negative dependency. Such programs may have multiple answer

sets, or none at all. In contrast, *stratified* programs have no cycles through negation and have a unique answer set.

The non-stratified dataset we consider, *coloring*, models probabilistic graphs with N nodes. Each node can be assigned one of three colors: red, green, or blue. An example in this dataset specifies a color assignment for all nodes and is labeled as either positive or negative, depending on whether it satisfies the constraints. Here, the edge probabilities are the parameters to be learned. This dataset was generated using a Python script in combination with clingo [115] to compute answer sets. An example of initial program is shown below.

```
red(X) :- node(X), \+ green(X), \+ blue(X).
blue(X) :- node(X), \+ green(X), \+ red(X).
green(X) :- node(X), \+ blue(X), \+ red(X).
qr :- e(A,B), red(A), red(B).
qr :- e(A,B), green(A), green(B).
qr :- e(A,B), blue(A), blue(B).
e(A,B):- edge(A,B).
e(A,B):- edge(B,A).
node(0..3).
t::edge(0,1).t::edge(0,2).t::edge(0,3).
t::edge(1,2).t::edge(1,3).t::edge(2,3).
query(qr).
```

The first three rules in the program ensure that a node X is assigned exactly one of the possible colors. The rules defining predicate qr identify conflicting color assignments, which occur when two connected nodes A and B share the same color. The atoms $edge(A, B)$ represent the presence of an edge between A and B for each pair of nodes $A, B \in \{0, \ldots, N-1\}$. To capture undirected edges, we use the predicate $e(A, B)$, which is defined in terms of the presence of either $edge(A, B)$ or $edge(B, A)$. The probabilities associated with the $edge/2$ atoms are tunable, and we indicate this using a placeholder, identified by t, instead of a numeric value. It is important to note that the same symbol is used for all such facts, but this does not imply that they are equal. The examples in the dataset were generated using predefined random target values for these tunable parameters. For each example, a random graph was created by including edges according to these fixed target probabilities. The resulting graph was

then used to generate the corresponding examples. This setup allows us to evaluate the effectiveness of the learning algorithm by comparing the learned probabilities against the original target values.

We used the MSE between learned and target values, denoted as MSE_{LT} (to differentiate it from the MSE defined in 8.6), to measure this deviation. MSE_{LT} is computed as follows:

$$MSE_{LT} = \frac{1}{N}\sum_{i=1}^{N}(\phi_i - t_i)^2$$

where ϕ_i and t_i are the learned and target probabilities of the $i-th$ parameter, and N is the number of tunable parameters.

Experiments were conducted on graphs with 4, 5, and 6 nodes. Below, we show an example consisting of two mega-examples: one positive and one negative.

```
#negative(1).
#atom(1,green(1)).
#atom(1,green(3)).
#atom(1,blue(0)).
#atom(1,red(2)).

#positive(2).
#atom(2,green(3)).
#atom(2,blue(2)).
#atom(2,red(0)).
#atom(2,red(1)).
```

We further evaluated our approach on the stratified *Bongard* dataset (see Example 8.3.1), to compare its performance against two state-of-the-art parameter learning solvers that operate on stratified programs: SLIPCOVER [72] and LIFTCOVER+ [116]. It is important to note that these tools do not support non-stratified programs, which distinguishes our method in terms of broader applicability. We generated datasets consisting of balanced sets of positive and negative examples, with 50, 100, and 250 examples. The underlying program used in this setting is shown in the snippet below.

```
pos :- r0, circle(A), inside(B,A).
pos :- r1, circle(A), triangle(B).
t::r0.
```

```
t::r1.
query(pos).
```

The first rule states that a shape B is inside a circle A, while the second states that a circle A and a triangle B are present. Atoms $r0$ and $r1$ are probabilistic facts whose probabilities, denoted by t, are to be learned.

Results

We evaluate PASP-GD in terms of log-likelihood (LL) and AUCROC.

Coloring dataset. For the coloring dataset, we evaluated the performance of our approach by measuring the final MSE_{LT} between the learned and the target probabilities. Additionally, we report the log-likelihood (LL) and AUCROC scores on the test set.

We tested different learning rate (lr) values (0.1, 0.5, and 0.9), using a maximum of 1000 iterations and randomly initialized parameters. Although the number of iterations required for convergence varied with the learning rate, the final results remained largely consistent across configurations. In all cases, convergence was typically reached well before the maximum number of iterations, usually within the first 100.

Table 8.3 summarizes the average AUCROC, LL, and MSE_{LT} scores obtained on the test set, across cross-validation folds, with learning rate set to 0.5 and a maximum of 100 iterations. PASP-GD achieves AUCROC scores ranging from 0.75 and 0.89,and MSE_{LT} often below 0.1. These results indicate that the learned probabilities are consistently close to the target values, demonstrating the effectiveness of the method in learning from the examples provided. For instance, with $N = 5$, AUCROC ranges from 0.8240 ($e = 50$) to 0.7530 ($e = 100$) before improving again to 0.8994 ($e = 250$), while LL ranges from -4.6855 ($e = 50$) to -11.9172 ($e = 100$) to -20.5940 ($e = 250$). A similar trend is visible for $N = 4$, but not for $N = 6$. This behavior may suggest that with $e = 100$, the model is better tuned for ranking predictions (thus optimizing AUCROC), but less well calibrated in terms of absolute probability values (LL) because the model has enough data to improve ranking performance (AUCROC) but not yet enough to fully optimize and calibrate probabilities, leading to suboptimal LL. With larger datasets ($e = 250$), the model benefits from improved calibration and overall

convergence. Overall, these results confirm the capability of our method to learn accurate probabilistic models in a graph-based domain.

Table 8.3: PASP-GD results on the coloring dataset with 5-fold cross-validation.

Nodes	Examples	Mean AUCROC	Mean LL	MSE_{LT}
4	50	0.8640	-5.1962	0.0313
4	100	0.8330	-14.5686	0.0488
4	250	0.8614	-23.4130	0.0019
5	50	0.8240	-4.6855	0.0550
5	100	0.7530	-11.9172	0.0937
5	250	0.8994	-20.5940	0.0373
6	50	0.8840	-8.0083	0.1182
6	100	0.8370	-12.2185	0.1371
6	250	0.8646	-22.9584	0.0937

Bongard dataset. We evaluated our approach on the Bongard dataset under several configurations, varying the learning rate (0.5, 0.9, and 1), the maximum number of iterations (set to 100), and the initialization strategy (fixed at 0.5 or randomized). As shown in Table 8.4, these variations had a negligible impact on performance: all configurations achieved the same AUCROC, with only minor fluctuations in LL, suggesting robustness to hyperparameter settings. In comparison with SLIPCOVER and LIFTCOVER+, our method consistently matched or outperformed both baselines in terms of LL and AUCROC across all configurations. With 50 examples, all algorithms achieved a LL around -3.7 and an AUCROC of 0.596. Increasing the number of examples to 100 led to the highest AUCROC. With 250 examples, both LL and AUCROC exhibited minor declines (around -19 and 0.7, respectively), but the results remained competitive.

Overall, this evaluation demonstrates that the proposed approach is effective on both stratified and non-stratified datasets, offering strong predictive performance and good generalization across varying data sizes and configurations.

8.4 Related Work

Many techniques exist for parameter learning in probabilistic logic programs, but most are restricted to programs with a unique model per world. PRISM [46] was among the

Table 8.4: PASP-GD results on the Bongard dataset with 5-fold cross-validation.

Algorithm	#Rules	#Examples	LR	Init. Prob.	Mean LL	Mean AUCROC
SLIPCOVER	2	50	-	-	-3.7938	0.5960
LIFTCOVER+	2	50	-	-	-3.7936	0.5960
PASP-GD	2	50	0.5	random	-3.7283	0.5960
PASP-GD	2	50	0.9	random	-3.7160	0.5960
PASP-GD	2	50	1.0	random	-3.7273	0.5960
PASP-GD	2	50	0.5	0.5	-3.7159	0.5960
PASP-GD	2	50	0.9	0.5	-3.7158	0.5960
PASP-GD	2	50	1.0	0.5	-3.7157	0.5960
SLIPCOVER	2	100	-	-	-6.7695	0.7390
LIFTCOVER+	2	100	-	-	-6.9500	0.7470
PASP-GD	2	100	0.5	random	-6.8277	0.7470
PASP-GD	2	100	0.9	random	-6.8278	0.7470
PASP-GD	2	100	1.0	random	-6.8278	0.7470
PASP-GD	2	100	0.9	0.5	-6.8278	0.7470
PASP-GD	2	100	0.5	0.5	-6.8277	0.7470
PASP-GD	2	100	1.0	0.5	-6.8278	0.7470
SLIPCOVER	2	250	-	-	-19.5835	0.6917
LIFTCOVER+	2	250	-	-	-19.9583	0.7000
PASP-GD	2	250	0.9	random	-19.3014	0.7112
PASP-GD	2	250	1.0	random	-19.3014	0.7112
PASP-GD	2	250	0.5	random	-19.3012	0.7112
PASP-GD	2	250	0.9	0.5	-19.3014	0.7112
PASP-GD	2	250	1.0	0.5	-19.3014	0.7112
PASP-GD	2	250	0.5	0.5	-19.3012	0.7112

first tools to address both inference and parameter learning in PLP, introducing an EM-based algorithm in its initial 1995 implementation. The same approach is adopted in EMBLEM [107] and in ProbLog2 [117], which learn the parameters of ProbLog programs from partial interpretations using the LFI-ProbLog algorithm [118]. LeProbLog [111] also targets ProbLog programs but solves the task via gradient descent.

Few tools handle PASP and allow multiple models per world. dPASP [119] performs parameter learning in Probabilistic Answer Set Programs under a different semantics, the max-ent semantics, where the probability of a query is the sum of the probabilities of the models where the query is true. They propose different algorithms based on the computation of a fixed point and gradient ascent. Parameter Learning under the Credal Semantics is available in PASTA [100, 101]. PASTA adopts the learning from

interpretations framework, but relies on an inference algorithm based on 2AMC, which has been empirically proven more effective [96], and on symbolic equation extraction, rather than projected answer set enumeration. This is also proved in our experimental evaluation.

Lastly, there are alternative semantics to adopt in the context of Probabilistic Answer Set Programming, namely, P-log [120], LPMLN [121], and smProbLog [106]. The relation among these has been partially explored by [122], though a comprehensive comparison is still lacking.

Part IV

Applications

Chapter 9

LIFTCOVER+ for Knowledge Graph Completion

A Knowledge Graph (KG) is a graph-based representation of the knowledge within a specific domain. KGs have gained popularity in the last decade ability to represent large and structured knowledge bases. Many types of real-world data can be naturally and effectively modeled as graphs, including computer networks, social networks, biomedical data, transportation systems, and more. For this reason, KGs are widely used in tasks such as query answering, recommender systems, chatbots, and voice assistants. Since KGs cannot contain all the knowledge relevant to a domain, they are typically incomplete and sparse. It is therefore often necessary to infer missing information, Knowledge Graph Completion (KGC) addresses this challenge: given a domain represented as a graph, the goal is to predict the missing edges among the entities involved.

9.1 Setting

A KG is composed of a set of triples (h, r, t), where h and t denote the head (start) and tail (end) entities, respectively, and r represents the relation linking them. Real-world KGs are usually incomplete and sparse, requiring the inference of missing information. This task is referred to as KGC. Depending on the information to be inferred, KGC can be divided into specific tasks [123], such as link prediction, entity prediction, or relation prediction.

Most state-of-the-art KGC methods rely on neural models, which, despite their strong performance, lack interpretability due to their black-box nature [124]. In contrast, PLP offers an interpretable framework, allowing for transparent and explainable models. Recently, the authors of [125] proposed learning logical rules from KGs and associating probabilistic weights to them.

Motivated by this line of research, we frame the KGC problem as a parameter learning task in PLP: given a set of probabilistic logical rules, the goal is to learn probabilities that maximize the likelihood of known triples in the KG. This formulation allows us to apply learning algorithms (e.g., Expectation Maximization) to KGC by treating observed triples as training data.

Link prediction is the task of predicting either the tail t of a triple $(h, r, ?)$, or the head h of a triple $(?, r, t)$. Formally, given a completion task $(h, r, ?)$ on a graph $\mathbb{G}$, the goal is to find an entity t such that $(h, r, t) \notin \mathbb{G}$. A KGC system assigns a score to each candidate triple, and candidates are ranked accordingly. For a prediction task $(h, r, ?)$, a candidate entity t is assigned a rank, denoted with $rank(t)$, based on its position in the list of scored triples. If multiple entities share the same score, tie-breaking strategies must be adopted. Common tie-breaking strategies include:

- Minimum ranking: assigns the lowest rank in the group.
- Maximum ranking: assigns the highest rank in the group.
- Average ranking: assigns the mean rank among tied elements.

In our implementation, we use average ranking, although other systems may adopt different strategies.

To evaluate link prediction performance, the original graph $\mathbb{G}$ is typically partitioned into three disjoint subsets: the training set T_{train}, test set T_{test}, and validation set T_{valid}, used to train the model and evaluate its performance. Given a set of test triples T_{test}, the following standard metrics are used to evaluate KGC models:

- Mean Rank (MR) is the average rank of answers to the test triples:

$$MR = \frac{1}{|T_{test}|} \sum_{t \in T_{test}} rank(t)$$

- Mean Reciprocal Rank (MRR) is the average reciprocal individual rank of the answers:

$$MRR = \frac{1}{|T_{test}|} \sum_{t \in T_{test}} \frac{1}{rank(t)}$$

- Hits@K represents the proportion of the answers ranked in the top K positions:

$$Hits@K = \frac{|\{t \in T_{test} \mid rank(t) \leq K\}|}{|T_{test}|}$$

In practice, the ranking for a test triple $(h, r, ?)$ may include candidate entities t' such that $(h, r, t') \notin T_{test}$. However, even though t' is not the correct answer, the triple (h, r, t') might be present in the training or validation sets. To avoid unfairly penalizing the correct prediction t, such triples are typically excluded from the ranking. This leads to the computation of filtered metrics, which more accurately reflect the model's performance [126].

9.2 Related Work

The KGC task has been extensively studied over the years, and a wide range of techniques has been proposed; a comprehensive survey can be found in [124]. Most approaches rely on deep learning, although notable exceptions exist, such as AnyBURL [125] and SAFRAN [127]. Another interesting method is proposed in [128], where the authors introduce "soft" rules integrated within a probabilistic model. They address the KGC task using EM with the Expectation step solved via belief propagation and a custom solution developed for the Maximization step, achieving competitive results compared to approaches like AMIE [129] and TransH [130].

A large body of work also exists on parameter learning in the Statistical Relational AI (StarAI) literature, particularly in PLP. Most methods rely on EM, such as [46, 101, 107, 117], though some exceptions exist [111, 131]. In this work, we focus on the liftable PLP system LIFTCOVER+, which allows inference to be performed much more efficiently than in traditional PLP. It should be noted, however, that the restrictions on clause types in liftable PLP may limit the language's expressiveness and make it harder to capture more complex relationships.

9.2.1 AnyBURL

AnyBURL [125] is a bottom-up rule learning algorithm designed to extract logical rules from KGs. It iteratively samples random paths from a KG $\mathbb{G}$, starting with paths of length $n = 2$ and progressively increasing the path length in subsequent iterations. Each sampled path is transformed into a ground logical rule, where the first edge in the path becomes the rule head and the remaining edges form the rule body.

Given a path composed of triples $\{(e_0, h_0, e_1), (e_1, b_1, e_2), \ldots, (e_n, b_n, e_{n+1})\}$, AnyBURL constructs a ground path rule R of the form:

$$h(e_0, e_1) \leftarrow b_1(e_1, e_2), \ldots, b_n(e_n, e_{n+1})$$

Based on the structure of these paths, AnyBURL extracts three distinct types of generalized rules:

- **C** rules are generalizations of cyclic ground path rules:
  $h(Y, X) \leftarrow b_1(X, A_2), \ldots, b_n(A_n, Y)$
- **AC1** rules are generalizations of both cyclic ($e_0 = e_{n+1}$) and acyclic ($e_0 \neq e_{n+1}$) ground path rules:
  $h(e_0, X) \leftarrow b_1(X, A_2), \ldots, b_n(A_n, e_{n+1})$
- **AC2** rules are generalizations of acyclic ground path rules:
  $h(e_0, X) \leftarrow b_1(X, A_2), \ldots, b_n(A_n, A_{n+1})$

where X and Y are variables that appear in the head, A_i are variables that can appear only in the body, and lowercase letters indicate constants. This rule discovery process continues until a saturation criterion is met, which depends on the marginal contribution of newly discovered rules to the model's overall performance.

Each rule R is assigned a confidence score, computed as follows [132]:

$$\mathit{support}(R) = |\{\theta_{XY} \mid \exists \theta_Z R_b \theta_{XYZ} \wedge R_h \theta_{XY}\}|$$

$$\mathit{conf}(R) = \frac{\mathit{support}(R)}{|\{\theta_{XY} \mid \exists \theta_Z R_b \theta_{XYZ}\}|}$$

where θ_{XY} is a grounding for variables X and Y appearing in the head R_h of R, θ_Z is a grounding for the variables appearing in the body R_b of R other than X and Y, and θ_{XYZ} is the union of θ_{XY} and θ_Z.

To score candidate entities, AnyBURL adopts the maximum aggregation strategy, in which candidate entities e_i are ordered according to the maximum confidence of all the rules R_i that generated them, i.e.,

$$score(e_i) = max\{conf(R_1), \ldots, conf(R_n)\}$$

where e_i is the entity and R_i are the rules that predicted it.

When the number of rules is high, computing KGC metrics is very expensive. To address this, AnyBURL introduces several optimizations, such as parallelizing rule application; truncating the grounding process to avoid excessive computation [133]; limiting the number of predictions returned for each query, resulting however in approximate rankings. Metrics computation is further accelerated through optimized libraries such as PyClause [134], which provides a native interface for AnyBURL.

9.2.2 SAFRAN

A known limitation of the maximum aggregation strategy used in systems like AnyBURL is that it relies solely on the single most confident rule to score a candidate entity, ignoring the contributions of the other rules that predict the same entity.

To address this, the Noisy-Or aggregation model has been proposed, which interprets rule confidences as probabilities and combines them under the assumption of conditional independence:

$$score(e) = 1 - \prod_{i=1}^{k}(1 - conf(R_i))$$

However, Noisy-Or suffers from a critical drawback: it does not account for rule redundancy. In real-world KGs, it is common for multiple rules to infer the same triple due to overlapping patterns. When such redundant rules are treated as independent, their combined effect leads to overestimation of confidence scores, ultimately degrading predictive performance [127]. To overcome this, the authors of [127] proposed SAFRAN, a framework introducing a refined aggregation method called *non-redundant Noisy-Or*. This approach clusters redundant rules based on their redundancy degree; then, it applies maximum aggregations to the rules in the same clusters; lastly, it aggregates predictions of different clusters with Noisy-Or. Two rules R_i, R_j are assigned

to the same cluster if their redundancy degree is higher than a certain threshold, that is $sim(R_i, R_j) > th$, where $sim(R_i, R_j)$ is the Jaccard Index of the sets of inferred triples and th is a threshold. To find the optimal value for th, which depends on the relation and rule type, SAFRAN adopts grid search or random search.

9.3 Approaches for Rule Generation

We considered three different strategies to generate rules whose probabilities are learned via LIFTCOVER+.

The first approach, which we refer to as *Paths*, focuses on generating cyclic rules by computing paths directly from the training triples in the knowledge graph. Each triple (h, r, t) is represented as an atom $t(h, r, t)$ We use predicate r/3 to represent both a relation and its inverse

```
r(A,r0,B):-r(A,r1,C),r(C,r2,D),r(D,r3,B).
```

where $r/3$ is defined as

```
r(S,R,T):-
  t(S,R,T).

r(S,i(R),T):-
  t(T,R,S).
```

Starting from these atoms, the system searches for connected paths of a given length, where each path corresponds to a potential rule structure. For instance, a sequence of relations $(r0, r1, r2, r3)$ is translated into a rule of the form

$$r(A, r0, B) : -r(A, r1, C), r(C, r2, D), r(D, r3, B).$$

Trivial or redundant rules are excluded.

The second approach is inspired by AnyBURL, and we call it *AC1 + AC2 + C*. For each relation in the KG, the method identifies two lists of start (head) and end (tail) nodes. Then, we find a user-defined number of paths starting from each start (end) node up to a user-defined length. These are used to generate **AC1**, **AC2**, and **C** rules, following AnyBURL's rule taxonomy. Duplicates are removed, and each rule

is initialized with a low probability (e.g., 10^{-4}). The resulting rule set is then used as input in LIFTCOVER+ to tune their probabilities. The rankings of the candidates for an entity are computed via Noisy-OR aggregation.

Finally, the *Weighted Sampling* approach generates rules by leveraging the frequency distribution of relations in the dataset. Each relation is assigned a sampling probability proportional to its occurrences, ensuring that more frequent relations are more likely to appear in the generated rules. A relation is first sampled as the head, while a random number of additional relations (between two and four) are sampled to form the body of the rule. These relations are then connected sequentially to create chain-like rules. Each generated rule is finally assigned a score proportional to the number of its body instantiations relative to the total across all rules. Unlike previous approaches, this one does not employ LIFTCOVER+ for parameter learning, as the rule scores are computed directly from structural statistics.

9.4 Experiments

Parameter learning experiments were conducted on the Leonardo HPC system of Cineca, using nodes equipped with an Intel Xeon Platinum 8358 processor (2.60 GHz, 32 cores), 481 GB of RAM, and NVIDIA A100 GPUs with 64 GB of memory. In contrast, the Weighted Sampling algorithm was executed on a machine featuring a 2.40 GHz CPU and 16 GB of RAM, with a maximum runtime of 8 hours.

9.4.1 Datasets

In our experimental evaluation, we employed four widely used benchmark datasets for KGC. FB15K-237 [135] is a subset of FreeBase comprising entity pairs and their corresponding relations. WN18RR [136] is derived from WordNet and consists of lexical entities connected through explicit semantic relations. NELL [137] originates from the Never-Ending Language Learner system, an autonomous agent designed to continuously extract and refine relational knowledge from web data. Finally, Nations [138] contains information on economic, diplomatic, military, and social interactions among 14 countries over the period 1950-1965. Table 9.1 summarizes, for each dataset, the number of triples in the training, validation, and test partitions, as well as the total number of relations and entities.

Table 9.1: For each dataset used in the experiments, we report the number of triples in the training, validation, and test sets, along with the total number of triples, relations, and entities.

Dataset	Train	Valid	Test	Total	Entities	Relations
Nations	1592	199	201	1992	28	55
WN18RR	86835	3034	3134	93003	71453	11
FB15k-237	272115	17535	20466	310116	14505	237
Nell	228426	129810	144307	502543	63361	400

9.4.2 Metrics computation

We implemented an algorithm in Prolog that, given a test set and a probabilistic theory, computes the standard metrics used for KGC. For each triple $(h, r, t) \in T_{test}$, the algorithm first computes the probability of the correct answer t according to the given theory. It then enumerates all possible instantiations of the query $r(h, r, T)$, representing alternative candidate entities $T = t'$. These instantiations are filtered to exclude any triples that also appear in the training or validation sets.

Next, the algorithm evaluates each remaining candidate t' by computing its probability under the theory. A counter N is maintained to track how many candidate answers have a probability greater than or equal to that of the correct answer t. Each such candidate is stored together with its probability, as it contributes to the final ranking of t. To improve computational efficiency, the process stops once 20 candidates with higher or equal probability have been found. This early stopping condition guarantees that the rank of the correct answer cannot be within the top 10, since even in the worst case, where all 20 answers have identical probability, their average rank would be $(20 + 1)/2 = 10.5$. Using this ranking information, the algorithm computes the exact values of the Hits@1, Hits@3, Hits@5, and Hits@10 metrics, differing from the metric computation of AnyBURL (also implemented in PyClause).

Due to early stopping, the estimated MRR may be slightly higher than the true MRR, since the algorithm considers at most 20 candidates with probability greater than or equal to that of the correct answer.

Table 9.2: Maximum length of the body of rules and number of rules for each dataset and each rule generation method.

Dataset	Rule Generation Method	Max Length	#Rules
Nations	Paths	3	305,365
	AC1 + AC2 + C	3	2,243
	Weighted Sampling	4	10^5
WN18RR	Paths	3	3,076
	Weighted Sampling	4	10^5
FB15K-237	Paths	3	33,696
	Weighted Sampling	4	10^5
Nell	Paths	3	101,242
	Weighted Sampling	4	10^5

9.4.3 Results

Table 9.2 reports, for each dataset, the maximum length of the rule bodies and the number of rules generated with the methods described in Section 9.3.

Table 9.3 presents the results of the experiments: for each dataset and algorithm, we report the metrics Hits@1, Hits@3, Hits@5, Hits@10, and MRR together with the time taken by parameter learning. The *Paths+Conf* approach indicates that the confidence values were set as parameters of the rules without performing EM. Also Weighted Sampling does not include an EM phase; thus, the reported metrics for this method are obtained solely from the initial sampling and weighting procedure described in Section 9.3. The confidence is computed using PyClause. From Table 9.3, we also observe that performing parameter learning with EM on the rules generated with Paths (denoted with *Paths+EM*) is not beneficial. Although this result may seem counterintuitive, since optimizing parameters to maximize log-likelihood is expected to yield a better classifier, it could be attributed to the necessity of subsampling positive examples and artificially generating negative examples. In such cases, filtering based on absence from training and validation sets does not guarantee true absence from the test set or overall falsity. Additionally, Weighted Sampling shows limited performance, being applicable to only two of the four datasets. This limitation is likely due to the large number of instantiations required to assign weights to the rules. Lastly, using AC1+AC2+C for rule generation and LIFTCOVER+ for parameter learning

Table 9.3: For each dataset and algorithm, we report the metrics Hits@1, Hits@3, Hits@5, Hits@10 and MRR together with the time taken by parameter learning. For Paths+Conf we used the confidence as the parameters of the rules. We do not report the configurations that were not able to solve the task.

Dataset	Approach	H@1	H@3	H@5	H@10	MRR	Time (s)
Nations	Paths+EM	0.4577	0.7861	0.8607	0.9950	0.6389	467.41
	Paths+Conf	0.4926	0.7910	0.8557	0.9701	0.6586	-
	AC1+AC2+C+EM	0.2189	0.5721	0.7612	0.9104	0.4404	6.69
	Weighted Sampling	0.5075	0.7811	0.8458	0.9652	0.6472	-
WN18RR	Paths+EM	0.3405	0.4537	0.4936	0.5378	0.4193	189.74
	Paths+Conf	0.4371	0.5057	0.5370	0.5731	0.4894	-
	AC1+AC2+C+EM	-	-	-	-	-	-
	Weighted Sampling	0.0306	0.0826	0.1107	0.1512	0.0741	-
FB15K-237	Paths+EM	0.2143	0.3093	0.3530	0.4263	0.2844	936.36
	Paths+Conf	0.2448	0.3528	0.4034	0.4801	0.3232	-
	AC1+AC2+C+EM	-	-	-	-	-	-
	Weighted Sampling	-	-	-	-	-	-
Nell	Paths+EM	0.0004	0.0018	0.0028	0.0053	0.0019	19,713.04
	Paths+Conf	0.0010	0.0025	0.0038	0.0067	0.0033	-
	AC1+AC2+C+EM	-	-	-	-	-	-
	Weighted Sampling	-	-	-	-	-	-

was not feasible for large datasets. Although this method works for smaller datasets like Nations, parameter learning on large datasets exceeds available memory resources.

To compute the metrics for the Paths approach, we employed PyClause using the Noisy-Or aggregation strategy, as it proved to be significantly faster than our Prolog implementation This outcome is somewhat surprising, given that Prolog systems such as SWI are supported by decades of software engineering expertise in improving the speed of answering this kind of query. One possible explanation is that the queries in our case are relatively simple and involve terms that are constants, which might cause the Prolog unification algorithm to incur excessive overhead because of its generality. Moreover, the implementation of PyClause in C++ and the several optimization strategies, while leading to approximate results, likely contribute to faster computation.

In summary, our empirical results show that parameter learning via EM on liftable probabilistic logic programs is currently not competitive for knowledge graph completion, especially due to scalability issues with large datasets.

Chapter 10

Application of LIFTCOVER+ to an Industrial Setting

We employed LIFTCOVER+ to support a financial company in optimizing interactions with its clients. Part of the work described in this chapter was carried out in collaboration with an industrial partner operating in the financial services sector. Due to commercial confidentiality, some technical and operational details cannot be disclosed in full. Therefore, the domain described in this chapter is a handcrafted scenario: the elements of the system presented here, such as the structure of the original data and the features, represent simplified, modified, or abstracted versions of those actually employed in the operational system. The context has also been adapted for confidentiality reasons. Nevertheless, the methodological principles and experimental considerations presented accurately reflect the approach and solutions that were effectively adopted.

The company's operators contact clients by phone to make financial proposals. The company required a system capable of assisting its operators in real time during these phone conversations by recommending the most appropriate proposal. For this purpose, we employed LIFTCOVER+.

All phone interactions are subject to a tagging process, in which each phone call is annotated with one or more tags describing what was said during the interaction, both by the operator and by the client. For instance, a tag is placed when clients ask for an installment plan or discount, or when they mention financial difficulties, when the operator makes a proposal. Each call is then labeled with an outcome, such as "Client Not Found", "Proposal Refused", or "Proposal Accepted".

The company provided data about its clients and contracts in the form of three tables extracted from its internal database, encompassing detailed information on debt collection cases and associated call interactions. The information on each debt collection case is stored in the CONTRACT table. The company provides the CONTRACT table already anonymized, with client-related information that does not allow for the identification of the individual. The INTERACTIONS table contains information on the phone interactions made by the company's operators. The TAG table contains all tags, in sequential order, extracted from the transcriptions of phone interactions, containing insights about the flow and development of the interaction between the client and the operator.

10.1 Data Preprocessing

Before applying the algorithms, a Python procedure processes the data and generates input files for the various algorithms. First, tags with a missing "Tag" field and interactions without tags are discarded. Incomplete or inconsistent records are handled, and textual fields from call transcripts are processed to extract meaningful information. For instance, a new feature regarding the mood of the phone call was created by extracting the corresponding information from the free-text field of the INTERACTIONS table. Numerical fields are discretized into categorical classes, since LIFTCOVER+ is designed to perform more effectively with categorical data. For example, the client's age was divided into the following classes: "<35", "34-45", "46-55", "56-65", "66-75", ">75", and "LegalEntity" (for legal entities). Special characters and accented letters are removed from all fields to prevent potential issues during algorithm execution. Finally, only cases, interactions, and tags of the field of interest are selected.

Then the processed dataset is randomly split into training and test sets, with 70% of the cases used to train the algorithm and the remaining 30% reserved for evaluation.

Only a subset of columns from each table was used. In particular, textual fields and columns with a high proportion of missing values were excluded to ensure data quality and computational efficiency. From the retained fields, logical predicates were generated to represent the information in a structured format suitable for LIFTCOVER+. For case-related fields, predicates take the form `field_name(IDContract, value)`, where IDContract identifies the corresponding case. Predicates related to the interactions and tags include both IDContract and IDInteraction: `field_name(IDContract,`

`IDInteraction, value)`. Additional auxiliary predicates were incorporated into the background knowledge to capture temporal relationships, such as the sequential order of interactions and tags.

Finally, the predicate `target_tag/1` was created as the target of the algorithm, representing the optimal action that the operator should take in their next interaction with the client.

10.2 Results

LIFTCOVER+ was employed for learning probabilistic rules aimed at predicting the operator's tags of interest, as identified by the company, in order to determine the most appropriate action the operator should take in the next call to achieve a positive outcome, i.e., a non-zero recovered debt amount. We selected LIFTCOVER+ not only for its strong predictive performance but also because its generated rules are easily interpretable and can be understood even by users without expertise in machine learning.

The company periodically provided new datasets containing information about additional cases and, in some instances, new attributes, as shown in Table 10.1. Datasets before Dataset_5 were not used for this study, and thus are not reported. For each dataset provided by the company, a new model was retrained on the combined set of new and existing data. As a result, several models of progressively increasing quality were produced, as both the dataset and the learning procedure were refined over time.

Table 10.1: Datasets used to train LIFTCOVER+. For each dataset, the new number of cases, interactions, and tags is reported, as well as new elements introduced.

Dataset	Notes	Cases (pos, neg)			Interactions	Tags
Dataset_5	New attributes	1936	(1550,	386)	6231	21040
Dataset_5_v2	New cases	349	(244,	105)	2464	2857
Dataset_6	New cases	122	(82,	40)	627	842
Dataset_7	New cases and target tags	209	(131,	78)	1659	1354
Dataset_8	New cases	327	(226,	101)	1983	2850
Dataset_9	New cases	619	(490,	129)	2967	5926
Dataset_10	New cases	136	(91,	45)	939	782

The rules induced by the algorithm represent the relationships between case infor-

mation, phone call records, and their associated tags, highlighting the operator actions that are most likely to result in a successful case outcome. A rule is expressed in the form: *head : p :- body* and can be interpreted as follows: if the body holds, then the head is true for a given case with probability *p*. Below are some examples of the rules learned by LIFTCOVER+:

- *target_tag(A, "solicit payment") : 0.871 :- offer_type(A, "investment"), region(A, "south").*

  This rule states that if the client comes from a southern region and the offer type is "investment", then the operator should solicit a payment from the client with a probability of 0.871.

- *target_tag(A, "propose offer") : 1.0 :- product(A, "leasing"), credit_score(A, "75-100").*

  This rule states that if the product is "leasing" and the client's credit score is between 75 and 100, then the operator should propose an offer to the client with certainty.

- *target_tag(A, "propose offer") : 0.532 :- offer_type(A, "utility contract"), clientAge(A, "25-30").*

  This rule states that if the offer refers to a utility contract and the client is between 25 and 30 years old, the operator should make an offer with a probability of 0.532.

In Table 10.2, the results obtained for various models trained with LIFTCOVER+ are reported in terms of AUCPR, AUCROC, and LL, where LL refers to the log-likelihood. Higher likelihood values indicate better performance, as an increase in likelihood means that the probability assigned by the model to positive examples increases, while that for negative examples decreases.

As can be observed from the results, model performance varies depending on the dataset, and in some cases may even worsen. This occurs because, as the number of examples increases, it may become more challenging to accurately represent the information conveyed by the examples. Therefore, a direct comparison between the model trained on Dataset_5, which exhibits the lowest LL, and the model trained on Dataset_10, the last one, cannot be entirely precise. It is also worth noting that, starting from the model trained on Dataset_7, the target labels of interest were modified.

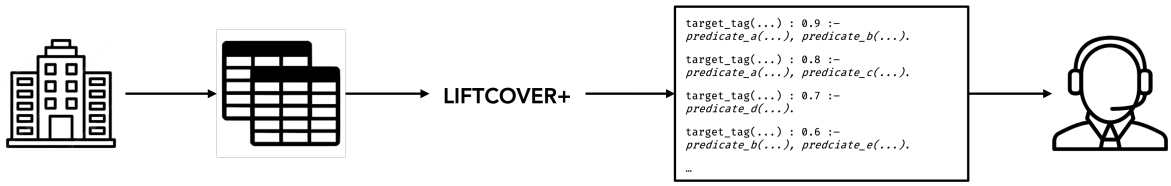


Fig. 10.1: Pipeline of this project, starting from the company, which provides the data. Data is used to build the predictive ILP system, and the output is used by the company operators.

Considering only the AUCPR and AUCROC values, later models still demonstrate good performance, with AUCPR showing a significant increase from Dataset_7 onward, while AUCROC decreases slightly.

Given the promising results, this system was incorporated into the pipeline employed by the company.

Table 10.2: Results obtained with LIFTCOVER+ in terms of AUCPR, AUCROC, and LL. For each dataset, the total number of cases, interactions, and tags (including both pre-existing and newly added records with that version) is reported.

Dataset	Cases (pos, neg)	Interactions	Tags	AUCPR	AUCROC	LL
Dataset_5	1936 (1550, 386)	6231	21040	0.912	0.681	-964.849
Dataset_5_v2	2285 (1794, 491)	2464	2857	0.901	0.711	-1924.943
Dataset_6	2407 (1876, 531)	627	842	0.885	0.814	-983.238
Dataset_7	2616 (2007, 609)	1659	1354	0.888	0.679	-1153.948
Dataset_8	2943 (2233, 710)	1983	2850	0.856	0.637	-1124.984
Dataset_9	3562 (2723, 839)	2967	5926	0.811	0.704	-1301.668
Dataset_10	3698 (2814, 884)	939	782	0.934	0.639	-2604.099

Chapter 11

Identifying risk factors of medication non-adherence via ML

11.1 Setting

Non-adherence to medication poses a significant obstacle to treatment effectiveness. This study is aimed at investigating the compliance to long-acting injection (LAIs) antipsychotics of the individuals affected by a psychosis spectrum disorder. The goal of the project is thus the development of a machine learning pipeline to determine whether subjects treated with LAIs are "compliant" or "non-compliant", based on their clinical and socio-demographic characteristics, and thus identify potential risk factors of medication non-adherence. Given the number of actual LAIs administrations (NA) and the number of expected LAIs administrations (NE), we define a subject as "compliant" if NA/NE ≥ 0.95, and "non-compliant" otherwise.

In Ferrara, Northern Italy, the Integrated Department of Mental Health and Pathological Addiction covers an area of 2630 km^2, with a catchment of 342,061 inhabitants as of 2020 [139]. Data for this study come from the FEPSY [140] database, which includes clinical and socio-demographic characteristics of the psychiatric patients of the province of Ferrara, northern Italy, who accessed outpatient mental health services in Ferrara during a period of 30 years (1991 – 2021). The FEPSY database has already been used for a similar project in [141]. For this study, we selected all the subjects who were treated with LAIs antipsychotics at least once, and with at least one diagnosis of psychotic spectrum disorders (International Classification of Diseases

- 9th revision (ICD-9) codes 295.xx, 297.xx, and 298.xx). The subset for this study includes 779 subjects, with 364 compliant subjects and 415 non-compliant. Table 11.1 and Table 11.2 respectively report the socio-demographic and clinical characteristics of the 779 individuals included in this study. For each characteristic, the statistical significance of the differences between groups was assessed using the appropriate test: t-test, Mann-Whitney U test, or chi-squared test, depending on the variable type.

11.2 ML Pipeline

Data Preprocessing and Feature Engineering Starting from the features described in Tables 11.1 and 11.2, we removed unnecessary or irrelevant features to reduce noise and improve model performance. In particular, binary variables with only one unique value and thus non-informative (e.g., Diagnosis ICD-9 code 302, 313, 315, Promazine Hydrochloride, and Antidepressants) were removed. Missing or unknown values were handled by treating them as a separate category. All features were standardized to ensure that variables were on the same scale, facilitating better convergence and model stability. We generated synthetic examples using the Synthetic Minority Oversampling Technique (SMOTE) [142] to increase the number of samples to 400 per class. Finally, feature selection was performed using a Random Forest model to identify the most informative subset of variables.

Model Selection Several ML models were selected to identify the most suitable one for this classification task, including: Decision Tree, Random Forest, AdaBoost, GradientBoosting, XGBoost [143], Support Vector Machine (SVM) [144], Logistic Regression, and Multilayer Perceptron (Neural Network). A 10-fold cross-validation strategy was employed to optimize hyperparameters for each model. This procedure involved testing multiple parameter combinations and selecting the configuration that yielded the best performance on the validation folds.

Model evaluation To ensure a robust assessment of model generalization and mitigate potential biases caused by a single train/test split, 10-fold cross-validation was also used during model evaluation. This approach provides a more reliable estimate of model performance across different subsets of the data. The impact of the predictors was evaluated using the coefficients from another logistic regression model, which

provides a measure of the strength and direction of each variable's contribution to the predicted outcome.

11.3 Experiments

All experiments were conducted on a cluster consisting of two machines with two AMD EPYC 9654 96-cores and four machines with two AMD EPYC 9454 48-cores. The memory limit was set to 100 GB, and the time limit was set to 10 hours. We used the *sklearn* [145] Python library for most ML models; the XGBoost model comes from [146].

The features selected by the Random Forest selector and used for the Table 11.4 presents the average performance metrics and standard deviations obtained by the evaluated models across the folds of cross-validation, while Figure 11.1 provides an overview of models' performance. Overall, the results indicate comparable performance among the models, although none of the algorithms tested achieved levels of accuracy or robustness sufficient to allow a reliable classification of compliant and non-compliant subjects. In particular, mean accuracy ranges from 56% to 61%, with Random Forest performing slightly better in this regard. The AUC ROC, which reflects the models' ability to distinguish between classes, with AdaBoost reaching 0.67. Figure 11.2 shows the ROC curves of each model across the folds. Notably, the recall for class 0 is particularly high in simpler models such as Logistic Regression (0.94) and Decision Tree (0.81), indicating that these models are effective at capturing instances of the class of interest, at the cost of slightly lower precision. Precision, however, remained more moderate (e.g., 0.55-0.63), highlighting that a portion of the identified cases may be false positives. In medical applications, a high recall is generally preferred, as failing to identify non-compliant or at-risk patients may have more serious consequences than unnecessarily treating compliant individuals. However, maintaining adequate precision is also critical, since allocating efforts or resources to already compliant patients would lead to an increase in costs. The F1-score, which balances precision and recall, ranged from 0.6 to 0.7, reflecting a reasonable trade-off between identifying the majority of non-compliant patients and limiting unnecessary interventions. Models such as Random Forest, GradientBoosting, and AdaBoost provide a balanced approach, capturing a substantial portion of non-compliant cases without excessively compromising precision.

Table 11.5 shows the coefficients, odds ratios (ORs), confidence intervals, and p-

values from a logistic regression model retrained on the entire dataset using only features with an importance greater than 0.005. These features were selected from the model that achieved the best average recall and that has the `feature_importance_` attribute, specifically the DecisionTree. Additionally, to address multicollinearity, the Variance Inflation Factor (VIF) was computed, and features with high VIF values were removed prior to model retraining. The logistic model, which comes from the *statsmodels* [147] Python library, differs from the logistic regression included among the various models tested in the pipeline, which was used for model comparison and evaluation. Coefficients were computed on the negative class, representing non-compliant patients. An $OR > 1$ corresponds to a higher probability of non-compliance and thus the corresponding feature may be a potential risk factor for LAIs non-compliance. Among the features, some catchment areas appear to be a prominent risk factor, along with disability, hospitalizations, and compulsory admissions. Marital status and age at first psychosis diagnosis may also contribute.

11.4 Discussion and Related Work

The low accuracy and overall modest performance metrics observed in our experiments can be attributed to several challenges inherent in the dataset and modeling process. First, the quality and quantity of the available data pose significant limitations. The dataset may lack sufficient examples to effectively train complex models, leading to underfitting and poor generalization to unseen cases. Additionally, the feature set may not fully capture the factors influencing patient compliance. Although standard preprocessing and feature selection were applied, different strategies and further improvements are needed to enhance the models' ability to discriminate between compliant and non-compliant patients.

The models we evaluated, along with the metrics used for assessment, align closely with the techniques commonly used for mental health prediction [148]. An overview of ML techniques for medication adherence was done by [149]. Authors of [150] also used XGBoost to predict medication adherence of patients with schizophrenia and attenuated psychotic disorders. In [151], ensemble learning and deep learning models were used to predict medical adherence of patients who self-administer injectable medication at home.

Beyond achieving strong predictive performance, there has been growing emphasis

on interpretable ML models, i.e., those that can be readily understood by humans [152]. Interpretability is particularly crucial in domains like healthcare [153,154], where trust in model decisions is essential. Three of the models we selected, namely logistic regression and decision tree, are recognized for their transparency and ease of comprehension even by non-specialists [152].

In conclusion, the pipeline demonstrates a systematic approach to developing a ML solution for compliant vs non-compliant patient classification. Although the results are not optimal, they represent a useful first step toward understanding the structure of the problem and defining more effective methodological strategies, providing a solid methodological foundation for subsequent, more in-depth analyses. Qualitative analysis of the most relevant variables suggests that certain clinical and socio-demographic features may play a key role in predicting compliance. By increasing the dataset size, improving feature engineering, and exploring alternative methods like ILP techniques, there is potential to improve both model performance and interpretability.

Table 11.1: Sociodemographic characteristics of the individuals included in this study.

Variable	Total ($N = 779$)	Compliant ($N = 364$)	Non-Compliant ($N = 415$)	*p-value*
Sex, n (%)				0.244
F	320 (41.1)	158 (43.4)	162 (39.0)	
M	459 (58.9)	206 (56.6)	253 (61.0)	
Nationality, n(%)				0.070
Italy	709 (91.0)	339 (93.1)	370 (89.2)	
Foreign	70 (9.0)	25 (6.9)	45 (10.8)	
Marital status, n(%)				0.227
Single	389 (49.9)	180 (49.5)	209 (50.4)	
Married or partnered	120 (15.4)	64 (17.6)	56 (13.5)	
Separated, divorced, or widowed	81 (10.4)	44 (12.1)	37 (8.9)	
Unknown	189 (24.3)	76 (20.9)	113 (27.2)	
Education, n(%)				0.057
Illiterate	34 (4.4)	18 (4.9)	16 (3.9)	
Literate	100 (12.8)	48 (13.2)	52 (12.5)	
Primary school	105 (13.5)	64 (17.6)	41 (9.9)	
Middle school	177 (22.7)	80 (22.0)	97 (23.4)	
High school	142 (18.2)	63 (17.3)	79 (19.0)	
College or university	24 (3.1)	8 (2.2)	16 (3.9)	
Unknown	197 (25.3)	83 (22.8)	114 (27.5)	
Occupational status, n(%)				0.777
Disabled	132 (16.9)	57 (15.7)	75 (18.1)	
Unemployed	120 (15.4)	56 (15.4)	64 (15.4)	
Employed	83 (10.7)	39 (10.7)	44 (10.6)	
Retired	67 (8.6)	36 (9.9)	31 (7.5)	
Other	32 (4.1)	12 (3.3)	20 (4.8)	
Homemaker	11 (1.4)	5 (1.4)	6 (1.4)	
Student	8 (1.0)	3 (0.8)	5 (1.2)	
Unknown	326 (41.8)	156 (42.9)	170 (41.0)	
Catchment area (district, n(%)				0.000
Ferrara	295 (37.9)	131 (36.0)	164 (39.5)	
Codigoro	173 (22.2)	86 (23.6)	87 (21.0)	
Cento	114 (14.6)	76 (20.9)	38 (9.2)	
Portomaggiore	110 (14.1)	32 (8.8)	78 (18.8)	
Copparo	56 (7.2)	34 (9.3)	22 (5.3)	
Unknown	31 (4.0)	5 (1.4)	26 (6.3)	

Table 11.2: Clinical characteristics of the individuals included in this study.

Variable	Total ($N = 779$)	Compliant ($N = 364$)	Non-Compliant ($N = 415$)	p-value
Age at first visit, mean (sd)	35.5 (12.1)	35.5 (12.2)	35.4 (12.0)	0.844
Age at first diagnosis, mean (sd)	39.2 (13.0)	39.2 (13.1)	39.1 (12.8)	0.989
First diagnosis ICD-9 code 295	407 (52.2)	185 (50.8)	222 (53.5)	
First diagnosis ICD-9 code 297	167 (21.4)	82 (22.5)	85 (20.5)	
First diagnosis ICD-9 code 298	205 (26.3)	97 (26.6)	108 (26.0)	
Age at first LAI, mean (sd)	46.6 (13.2)	47.2 (13.3)	46.0 (13.2)	0.201
Days between first diagnosis and first LAI	4074.0 (2832.2)	4295.5 (2773.3)	3879.8 (2872.2)	0.057
LAI Type, n (%)				
Haloperidol	7 (0.9)	1 (0.3)	6 (1.4)	0.178
Haloperidol decanoate	435 (55.8)	204 (56.0)	231 (55.7)	0.972
Aripiprazole	42 (5.4)	23 (6.3)	19 (4.6)	0.361
Fluphenazine decanoate	146 (18.7)	66 (18.1)	80 (19.3)	0.752
Olanzapine	1 (0.1)	0 (0.0)	1 (0.2)	1.000
Olanzapine pamoate monohydrate	1 (0.1)	0 (0.0)	1 (0.2)	1.000
Paliperidone palmitate	195 (25.0)	88 (24.2)	107 (25.8)	0.664
Promazine Hydrochloride	0 (0.0)	0 (0.0)	0 (0.0)	NA
Risperidone	192 (24.6)	76 (20.9)	116 (28.0)	0.028
Zuclopenthixol acetate	1 (0.1)	1 (0.3)	0 (0.0)	0.948
Zuclopenthixol decanoate	43 (5.5)	18 (4.9)	25 (6.0)	0.617
Comorbidity, n (%)				
Diagnosis ICD-9 code 290	10 (1.3)	5 (1.4)	5 (1.2)	1.000
Diagnosis ICD-9 code 291	9 (1.2)	4 (1.1)	5 (1.2)	1.000
Diagnosis ICD-9 code 292	20 (2.6)	8 (2.2)	12 (2.9)	0.701
Diagnosis ICD-9 code 293	10 (1.3)	3 (0.8)	7 (1.7)	0.454
Diagnosis ICD-9 code 294	13 (1.7)	6 (1.6)	7 (1.7)	1.000
Diagnosis ICD-9 code 296	168 (21.6)	82 (22.5)	86 (20.7)	0.600
Diagnosis ICD-9 code 299	12 (1.5)	5 (1.4)	7 (1.7)	0.950
Diagnosis ICD-9 code 300	134 (17.2)	69 (19.0)	65 (15.7)	0.263
Diagnosis ICD-9 code 301	190 (24.4)	78 (21.4)	112 (27.0)	0.086
Diagnosis ICD-9 code 302	0 (0.0)	0 (0.0)	0 (0.0)	NA
Diagnosis ICD-9 code 303	28 (3.6)	9 (2.5)	19 (4.6)	0.167
Diagnosis ICD-9 code 304	25 (3.2)	6 (1.6)	19 (4.6)	0.035
Diagnosis ICD-9 code 305	41 (5.3)	18 (4.9)	23 (5.5)	0.832
Diagnosis ICD-9 code 306	1 (0.1)	0 (0.0)	1 (0.2)	1.000
Diagnosis ICD-9 code 307	10 (1.3)	6 (1.6)	4 (1.0)	0.598
Diagnosis ICD-9 code 308	9 (1.2)	7 (1.9)	2 (0.5)	0.123
Diagnosis ICD-9 code 309	60 (7.7)	27 (7.4)	33 (8.0)	0.885
Diagnosis ICD-9 code 310	1 (0.1)	0 (0.0)	1 (0.2)	1.000
Diagnosis ICD-9 code 311	7 (0.9)	5 (1.4)	2 (0.5)	0.350
Diagnosis ICD-9 code 312	6 (0.8)	0 (0.0)	6 (1.4)	0.058
Diagnosis ICD-9 code 313	0 (0.0)	0 (0.0)	0 (0.0)	NA
Diagnosis ICD-9 code 314	1 (0.1)	0 (0.0)	1 (0.2)	1.000
Diagnosis ICD-9 code 315	0 (0.0)	0 (0.0)	0 (0.0)	NA
Diagnosis ICD-9 code 316	1 (0.1)	1 (0.3)	0 (0.0)	0.948
Diagnosis ICD-9 code 317	28 (3.6)	11 (3.0)	17 (4.1)	0.541
Diagnosis ICD-9 code 318	15 (1.9)	7 (1.9)	8 (1.9)	1.000
Diagnosis ICD-9 code 319	9 (1.2)	4 (1.1)	5 (1.2)	1.000
Medication, n (%)				
Antidepressants	0 (0.0)	0 (0.0)	0 (0.0)	NA
Oral Antipsychotics	644 (82.7)	289 (79.4)	355 (85.5)	0.030
Clozapine	53 (6.8)	21 (5.8)	32 (7.7)	0.352
Mood Stabilizers	174 (22.3)	68 (18.7)	106 (25.5)	0.027
Benzodiazepines	610 (78.3)	275 (75.5)	335 (80.7)	0.097
Other medications	425 (54.6)	207 (56.9)	218 (52.5)	0.254
No. hospitalizations, mean (sd)	5.6 (11.7)	4.8 (11.3)	6.3 (12.0)	0.000
No. hosp. before LAI, mean (sd)	3.6 (7.2)	3.3 (7.0)	3.8 (7.4)	0.088
No. hosp. after LAI, mean (sd)	2.0 (6.2)	1.4 (4.9)	2.5 (7.0)	0.000
Duration of hospitalization, mean (sd)	83.5 (141.3)	75.3 (144.7)	90.7 (138.0)	0.002
Duration of hosp. before LAI, mean (sd)	52.9 (90.7)	51.8 (98.5)	53.8 (83.3)	0.217
Duration of hosp. after LAI, mean (sd)	30.6 (77.5)	23.5 (69.0)	36.9 (83.8)	0.000
Compulsory admissions, mean (sd)	1.0 (2.1)	0.8 (2.1)	1.1 (2.1)	0.001
Comp. adm. before LAI, mean (sd)	0.6 (1.1)	0.5 (1.0)	0.6 (1.2)	0.152
Comp. adm. after LAI, mean (sd)	0.4 (1.5)	0.3 (1.6)	0.5 (1.5)	0.000

Feature	Importance
Days between first diagnosis and first LAI	0.072
Age at first diagnosis	0.062
Age at first visit	0.060
Age at first LAI	0.058
Duration of hospitalization	0.055
Duration of hosp. before LAI	0.050
No. hospitalizations	0.037
Duration of hosp. after LAI	0.036
No. hosp. before LAI	0.032
No. hosp. after LAI	0.027
Compulsory admissions	0.020
SPT_Cento	0.018
Comp. adm. before LAI	0.016
Other medications	0.016
SPT_Ferrara	0.015
SPT_Portomaggiore	0.014
Occupational status_Unknown	0.013
Haloperidol decanoate	0.013
First diagnosis = ICD-9 code 295	0.012
Paliperidone palmitate	0.012
Education_PrimarySD	0.012
Sex_M	0.012
Comp. adm. after LAI	0.012
Risperidone	0.011
SPT_Codigoro	0.011
Sex_F	0.011
Occupational status_Disabled	0.011
Diagnosis ICD-9 code 301	0.011
Diagnosis ICD-9 code 296	0.010
Marital status_Single	0.010
Diagnosis ICD-9 code 300	0.010
Benzodiazepines	0.010
First diagnosis ICD-9 code 298	0.010
Oral Antipsychotics	0.010
SPT_Unknown	0.010
Marital status_Married or partnered	0.010
Education_Unknown	0.010
Marital status_Unknown	0.009
Mood Stabilizers	0.009
First diagnosis ICD-9 code 297	0.009
Fluphenazine decanoate	0.009
Education_High school	0.009
Marital status_Separated, divorced, or widowed	0.009
Education_Middle school	0.009
Occupational status_Unemployed	0.008

Table 11.3: Feature importances of the predictors included in the evaluated machine learning models, computed using a Random Forest as a feature selector. Higher values indicate greater influence on the models' predictions.

Model	Accuracy	Precision	Recall	F1-score	AUC ROC
AdaBoost	0.601 (0.052)	0.629 (0.048)	0.607 (0.089)	0.616 (0.063)	0.666 (0.056)
DecisionTree	0.564 (0.030)	0.564 (0.025)	0.812 (0.122)	0.662 (0.041)	0.665 (0.068)
GradientBoosting	0.601 (0.094)	0.625 (0.088)	0.612 (0.132)	0.616 (0.107)	0.601 (0.035)
LogisticRegression	0.561 (0.040)	0.552 (0.024)	0.944 (0.058)	0.696 (0.029)	0.644 (0.098)
MLP	0.566 (0.066)	0.592 (0.063)	0.598 (0.086)	0.593 (0.069)	0.605 (0.065)
RandomForest	0.606 (0.061)	0.623 (0.053)	0.658 (0.087)	0.639 (0.063)	0.606 (0.057)
SVM	0.584 (0.037)	0.572 (0.026)	0.879 (0.053)	0.692 (0.029)	0.646 (0.060)
XGBoost	0.592 (0.083)	0.621 (0.079)	0.593 (0.118)	0.604 (0.093)	0.607 (0.061)

Table 11.4: Average metrics (and standard deviations) of the models across the 5 folds of cross-validation.

Feature	Coefficient	Odds Ratio	95% CI	*p-value*
SPT_Unknown	1.702	5.485	[1.996, 15.072]	0.001
SPT_Portomaggiore	1.044	2.840	[1.721, 4.684]	0.000
Occupational status_Disabled	0.380	1.463	[0.977, 2.188]	0.064
SPT_Ferrara	0.357	1.429	[1.000, 2.041]	0.050
No. hosp. after LAI	0.257	1.293	[1.017, 1.646]	0.036
Compulsory admissions	0.130	1.139	[0.886, 1.463]	0.309
Marital status_Single	0.104	1.109	[0.810, 1.519]	0.519
Age at first diagnosis	0.054	1.056	[0.900, 1.237]	0.505
Comp. adm. before LAI	0.014	1.014	[0.806, 1.275]	0.905
Days between first diagnosis and first LAI	-0.069	0.933	[0.797, 1.092]	0.389
Duration of hosp. before LAI	-0.075	0.928	[0.774, 1.111]	0.415
Intercept	-0.135	0.873	[0.639, 1.194]	0.396
Other medications	-0.144	0.866	[0.742, 1.010]	0.067
SPT_Cento	-0.610	0.543	[0.336, 0.879]	0.013
Education_Primary school	-0.647	0.524	[0.336, 0.816]	0.004

Table 11.5: Coefficients, odds ratio (OR), confidence intervals, and p-values of the logistic regression model computed for class 0. $ORs > 1$ indicate features increase the probability of class 0, and can therefore be considered risk factors for LAI non-compliance; $ORs < 1$ indicate the opposite effect. Note that this is not the same logistic regression included in the pipeline, but one used specifically to extract coefficients.

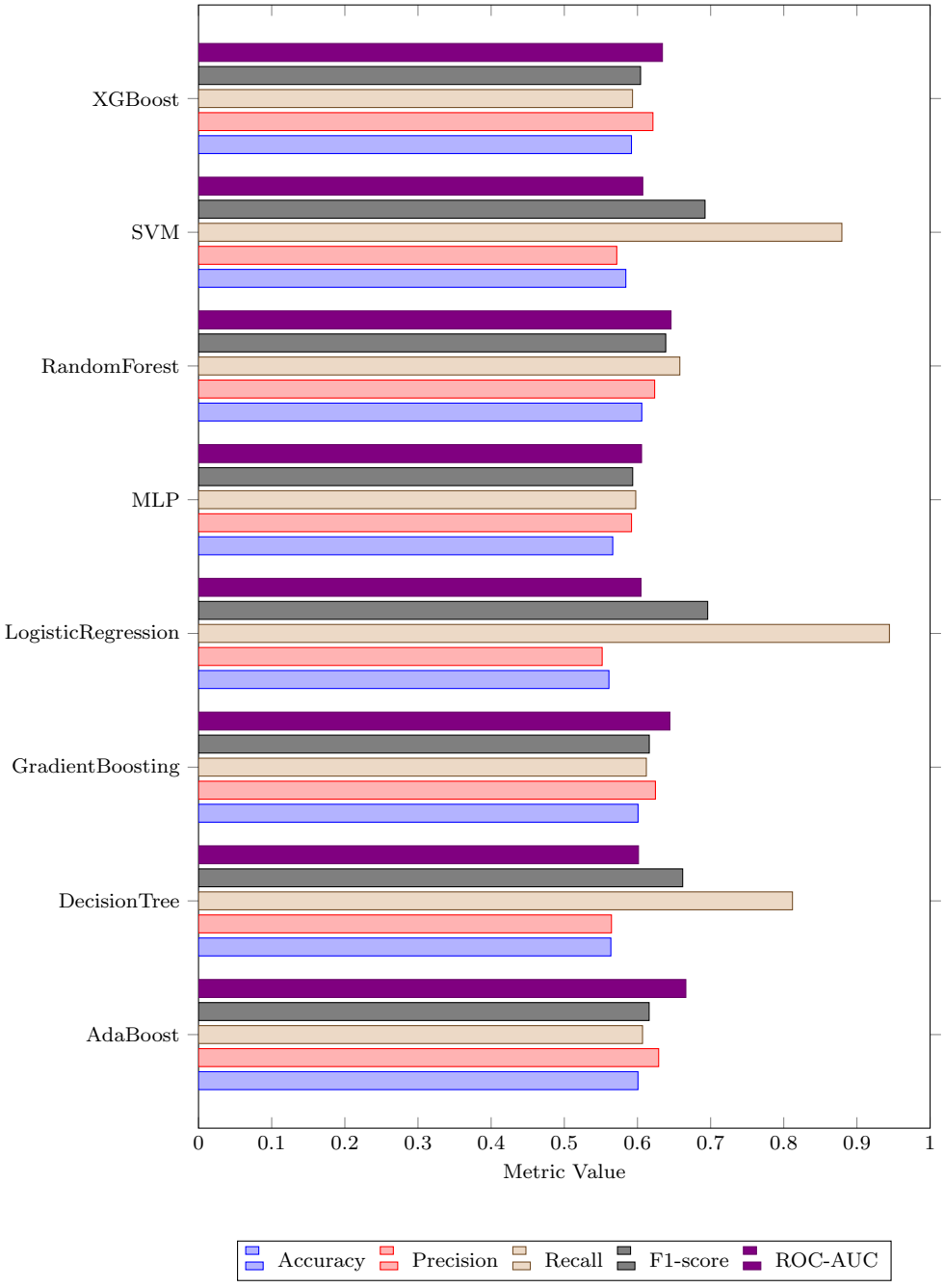


Fig. 11.1: Comparison of models' performance. Values are mean metrics over cross-validation folds, calculated considering class 0 as the class of interest.

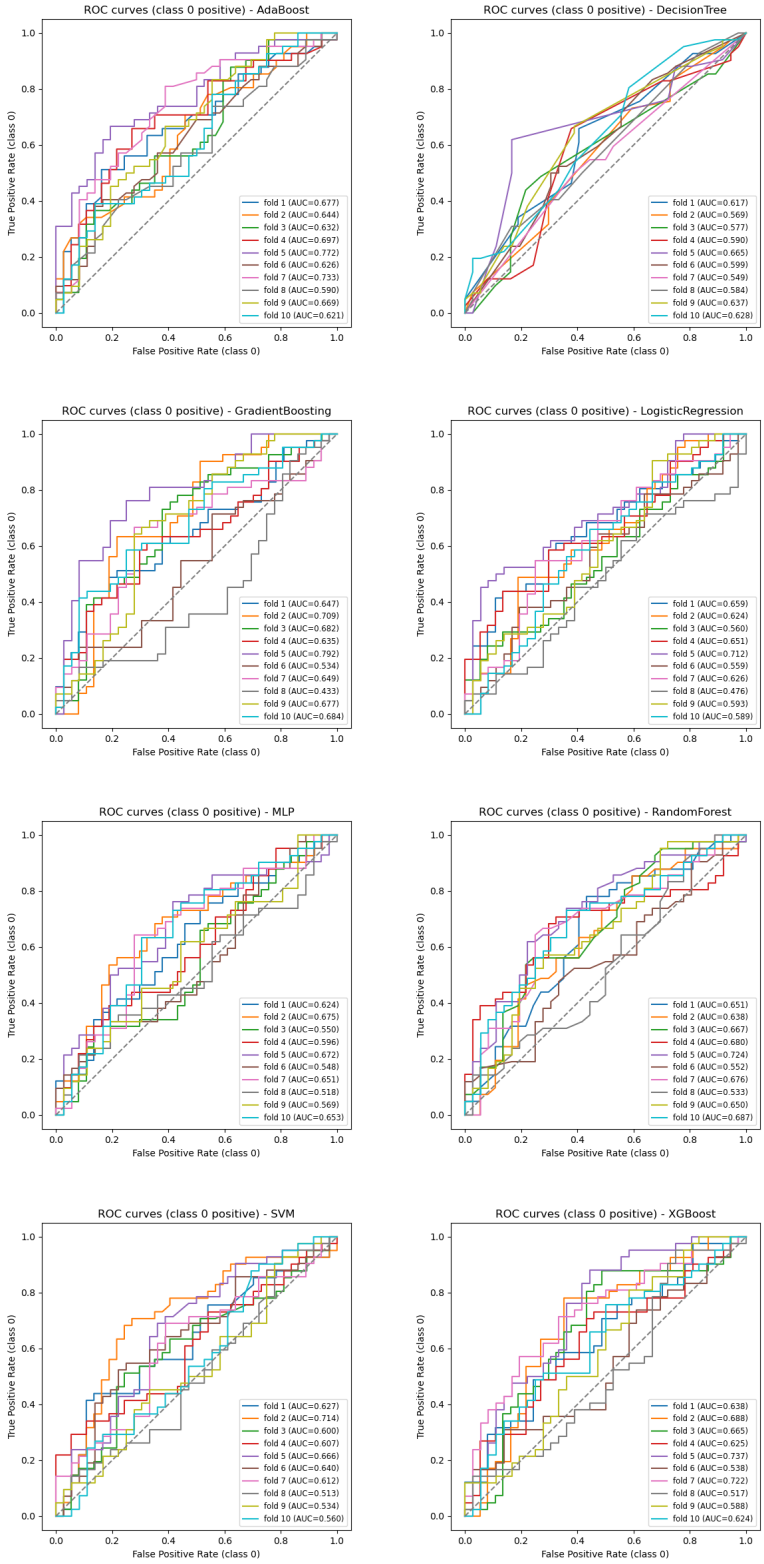


Fig. 11.2: AUC-ROC curves for each model across all cross-validation folds.

Chapter 12

NeSy Benchmark

Neuro-Symbolic AI still lacks standardized benchmarks, which prevents objective advancement assessment and slows the development of more effective and scalable methods. Creating such benchmarks is therefore essential to support research efforts and facilitate the practical deployment in this field.

To help bridge this gap, we contributed two datasets based on the tic-tac-toe game: one for identifying the winner of a complete board and another for choosing the best next move to play. Both datasets[1] are part of a broader initiative to establish a foundational benchmark suite for NeSy systems, alongside datasets like CLEVRHans [155], Kandinsky Patterns [156], and CLE4EVR [157].

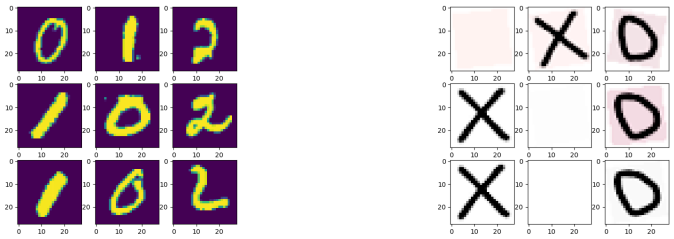
12.1 Dataset Generation

12.1.1 Identifying the winner

The first dataset consists of all possible valid Tic-Tac-Toe board configurations, where either player X wins, player O wins, or the game ends in a draw. We used the Python code shown below to generate these board configurations, exploring all potential endgame states.

```
def generate_random_board():
    # X starts, alternating X and O
    symbols = ['X', 'O'] * 4 + ['X']
```

[1] Available at https://github.com/bizzarriA/TicTacToeDS



(a) Label: o (b) Label: o

Fig. 12.1: Examples of board generated with MNIST digits (a) and with handwritten symbols (b). In both cases, the label `o` indicates the winner of the game.

```
    # Empty 3x3 board
    board = [['b'] * 3 for _ in range(3)]
    # Shuffle to explore different move sequences
    random.shuffle(symbols)

    for i in range(9):
        row, col = divmod(i, 3)
        board[row][col] = symbols.pop()
        if winner := check_winner(board):
            # Stop if a winner is found
            return board, winner

    # Return 'b' for a draw
    return board, 'b' if valid_game(board) else None

def valid_game(board):
    count_x = sum(row.count('X') for row in board)
    count_o = sum(row.count('O') for row in board)
    # Ensure a valid state
    return count_x == count_o or count_x == count_o + 1
```

An example is thus a complete board composed of nine images, of either MNIST digits or handwritten symbols. The label is the winner of the game, and we use images of `0` or nothing for blank cells, `x` or 1 for player X, and `o` or 2 for player O. Figure 12.1 shows examples of such boards.

12.1.2 Choosing the next move

The second dataset focuses on the identification of the best next move to make, given an incomplete board. In this case, the label is the position on the board of the cell in which the current player should place their symbol. This dataset presents a more complex challenge, requiring the model to predict the optimal strategic move given the current board state. To generate random board configurations and identify the corresponding next move, we used the following Prolog code, which shows the key functions and logic involved (note that this is a simplified excerpt, not the full implementation[2]):

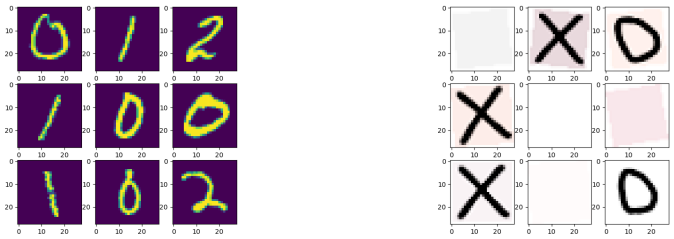
```
% Prolog code snippet for generating the next move dataset
generate_random_board(Board):-
    length(Board,9),
    place_elements(Board),
    valid(Board), !.
generate_random_board(Board):-
    generate_random_board(Board).

% Determine the next move
next(Column,Row):-player(Player),line(Column,Row,Player).
next(Column,Row):-player(Player),opponent(Player,Opponent),
line(Column,Row,Opponent).

pair_(Board,L):-
    generate_random_board(Board),
    assert_board(Board),
    pair(Board,L),
    retractall(cell(_,_,_)), !.
pair_(_,_):-
    retract(player(_)),
    retractall(cell(_,_,_)).

pair(Board,[Column,Row]):-
    next_player(Board,Player),
```

[2]Available at https://github.com/bizzarriA/TicTacToeDS



(a) Label: (3,2) (b) Label: (3,2)

Fig. 12.2: Examples of board generated with MNIST digits (a) and with handwritten symbols (b). In both cases, the label `o` indicates the winner of the game.

```
    opponent(Player,Opponent),
    assertz(player(Player)),
    \+win(Player),\+win(Opponent),
    \+full_board,
    next(Column,Row),
    retract(player(Player)).
pair(Board,[]):-
    writeln(Board),
    win(Player),
    display_game(Board).
```

For example, given the board in Figure 12.2, the current player (O) should place their symbol in the third cell of the second row to win the game, and thus the label is (3,2).

By integrating board configurations and visual data, we created comprehensive datasets that challenge models to simultaneously develop visual perception skills (identifying board symbols) and reasoning abilities (predicting the game outcome or next move).

Chapter 13

Predicate Renaming via Large Language Models

In the last decades, many ILP systems have been successfully applied [65] to different domains, such as bioinformatics and drug design [158–160], natural language processing [161], games [162,163], data curation and transformation [164], explainable AI [165], and learning from trajectories [166].

Some ILP systems are capable of performing Predicate Invention (PI), i.e., introducing new unnamed predicates into the learned program to adhere to a language bias or to create simpler rules. Examples of such systems are POPPI [167], PHIL [168], Nesy-π [169], and many others. The presence of such predicates poses a big challenge for both programmers and final users, since it is difficult to understand their intended meaning, especially when the output theory is composed of a large number of rules. Additionally, manually renaming them is usually unfeasible, since it requires the availability of a domain expert. In fact, unnamed predicates not only complicate human understanding of the logic program but also hinder the debugging process for programmers.

We thus propose to use LLMs to provide names to these predicates. The goal of this work is to exploit LLMs to generate semantically meaningful predicate names, which should be naturally understood by the user. To the best of our knowledge, there are no systems that perform automatic renaming of unnamed predicates, and this is the first study trying to address this issue.

13.1 Predicate Invention

ILP systems may sometimes introduce invented predicates that are not explicitly defined in the knowledge base or problem description. There are many reasons to invent new predicates, such as capturing intermediate relationships, simplifying complex rules, enhancing the program's modularity and flexibility, expressing a new concept, improving the accuracy while reducing the complexity of a hypothesis, or when the given vocabulary is insufficient to express the target theory. However, invented symbols have a placeholder name, such as "*inv*". The lack of a meaningful name makes it difficult to understand the intended use of the predicate and its reusability.

In ILP [170], PI was first introduced in inverse resolution [171], and early works have been summarized in [172]. PI refers to those techniques that enable systems to automatically discover new predicates, which are not part of the initial knowledge base [173]. One of the most straightforward examples of PI, shown in Example 13.1.1, is the invention of the predicate `parent` to learn the definition of `grandparent`, given only the predicates `mother` and `father`:

Example 13.1.1. Example of PI, where the invented predicate `inv` represents the parent relation.

```
grandparent(A,B) :- inv(A,C),inv(C,B).
inv(A,B) :- mother(A,B).
inv(A,B) :- father(A,B).
```

In this way, a more compact theory has been created. Despite gaining increasing attention in recent developments, PI is still not widely spread [65]. As a matter of fact, the invention of new, useful predicates is not a trivial task [174], as it is often challenging to determine when and if they are necessary, how many arguments they should have (i.e., their arity), and what their order and type should be [173, 175].

METAGOL [176] is a metarule-based approach that seeks to overcome this limitation by using metarules to determine and restrict the syntax of rules, although they should be predefined by the user.

POPPER [177] is an ILP system based on the Learning From Failures (LFF) framework, which decomposes the ILP problem into three steps: generate, test, and constrain. In each cycle, POPPER generates a logic program based on current con-

straints, evaluates it against examples, and, if the hypothesis is incomplete or inconsistent, derives new constraints from the failure. This iterative refinement continues until a consistent and complete program is found. POPPI [167] is an ILP method that supports the creation of new predicates without needing human intervention to specify the arity or metarules. POPPI thus extends POPPER by enabling automatic PI, casting the ILP problem as an Answer Set Programming (ASP) task.

The authors of [169] proposed NeSy-π, a NeSy approach for the classification of complex visual scenes, that processes visual inputs with deep neural networks and invents new concepts.

Nevertheless, the discovery of new predicates is not limited to PI. In the context of PLP [178], HPLPs [179] are a restriction of Logic Programs with Annotated Disjunctions (LPADs) [53]. In HPLPs, clauses and predicates are hierarchically organized. Parameter learning for Hierarchical Probabilistic Logic Programs (PHIL) [168] is an algorithm for learning the parameters of HPLPs. Given a specific form of LPADs defining the target predicate r in terms of input predicates, for which the definition is given as input and is certain, and hidden predicates, defined by the rules of the program, PHIL learns the probabilities of the program. Each clause in the program has a single atom in the head annotated with a probability. These programs can be divided into layers. Each layer defines a set of hidden unnamed predicates in terms of either the predicates from the layer below or the input predicates. In Example 13.1.2 from [168], about the UWCSE domain [180], we can see an example of such predicates. The first clause, C_1, states that the probability of a student (A) being advised by a professor (B) is 0.3 if both A and B have worked on a project (C), and there is a relation r_1_1 between the student, the professor, and the project. Relation r_1_1 defined by clause $C\_1\_1\_1$, which states that r_1_1 holds with a probability of 0.2 if there is a publication P authored by both A and B and produced within project C. r_1_1 can then be renamed as "collaboratedOn". We can thus say that a student (A) is advised by a professor (B) if there is a project (C) on which both have worked, and A and B have collaborated on project C.

Clause C_2 states that a student (A) is advised by a professor (B) with a probability of 0.6 if A is a teaching assistant (ta) for course C, which is taught by B.

Example 13.1.2. Unnamed predicate r_1_1 created by PHIL in the learning process.

```
C_1 = advised_by(A,B) : 0.3 :- student(A), professor(B),
```

```
   project(C,A), project(C,B), r_1_1(A,B,C).
C_2 = advised_by(A,B) : 0.6 :- student(A), professor(B), ta
   (C,A), taughtby(C,B).
C_1_1_1 = r_1_1(A,B,C) : 0.2 :- publication(P,A,C),
   publication(P,B,C).
```

In the context of abductive logic programming [181], the approach proposed by [182] aims to abduce relations by leveraging linear algebra in vector spaces. Hidden relationships between entities are learnt through matrix operations, making it useful for applications where explicit rules are hard to define. Focusing on Datalog programs with binary predicates, the authors address both non-recursive and recursive cases. For instance, in the non-recursive case, given two relations $r_1(X, Y)$ and $r_3(X, Z)$, represented as adjacency matrices $\mathbf{R}_1$ and $\mathbf{R}_3$ respectively, the system learns a new relation $r_2(Y, Z)$, represented by matrix $\mathbf{R}_2$, that explains how r_1 and r_3 interact. This is done by solving the matrix equation $\mathbf{R}_3 = min_1(\mathbf{R}_1\mathbf{R}_2)$, where $min_1(x)$ is a nonlinear function, so that $r_3(X, Z) \Leftrightarrow \exists Y r_1(X, Y) \wedge r_2(Y, Z)$. Although r_1 and r_3 are known, r_2 is not, therefore it could be useful to find a meaningful name for easier reuse. For example, the authors have applied this method to a subset of the FB15k [126] knowledge graph, involving person and film. By considering the relations *language(X,Z)* and *genre(X, Y)*, the following rule can be discovered:

$$language(X, Z) \Leftrightarrow genre(X, Y) \wedge r_{2\_abd}(Y, Z).$$

Here, the abduced relation r_{2_abd}, which isn't originally in the graph, could be interpreted as *genre_lang*, implying that genre and language are somewhat related to each other. However, such an interpretation on the new predicate r_{2_abd} as well as the predicate name *genre_lang* have to be given by the user.

13.2 Renaming predicates using LLMs

In this section, we outline our approach for assigning meaningful names to unnamed predicates in a set of logical rules.

Figure 13.1 depicts the pipeline we propose. After deciding the number n of LLMs to query for suggestions, the number k of suggestions they should provide, and the

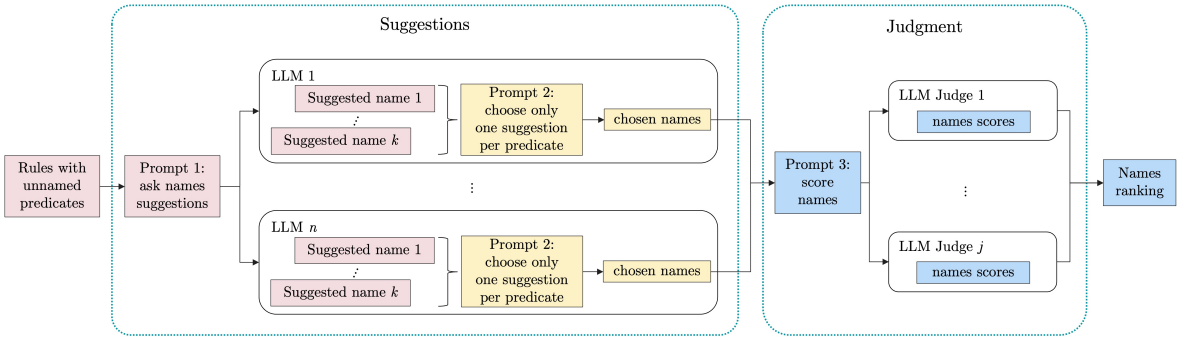


Fig. 13.1: Diagram illustrating key components and interactions of the approach we propose.

number j of LLM judges, the pipeline consists of three main steps:

1. Asking the n LLMs to find a meaningful name for each unnamed predicate, for k times to tackle possible hallucinations; for these preliminary experiments, we decided to repeat the question three times.

2. Asking the models to choose the most suitable name among their own suggested names.

3. Asking the j judges to score the different proposed names, obtaining a ranking.

13.2.1 Materials

To assess the ability of the LLMs to find correct names for unnamed predicates, we created different examples of logic theories, which represent the possible output of an ILP system in a declarative logic programming language. Although constructs that are specific to Prolog were used for these experiments, this approach could be applied to scenarios where other logic programming languages are employed. For example, if rules are written in a purely declarative manner, those rules may be commonly used for both Prolog and ASP.

In each set of rules, we replaced the true names of some or all predicates with placeholders. We tested various formats (e.g., A, P, h0, and so on) for placeholders to determine whether the placeholder name influences the suggestions. Unnamed predicates can occur in the head and/or in the body of one or multiple rules. For all case studies presented in this paper, the true names of the unnamed predicates are included

as comments (e.g., `# P = coauthors`) for the reader's reference. These comments were not included in the prompts provided to the LLMs.

Unnamed predicates appearing only in the head The first and simplest case study, *coauthors*, consists of a single rule, as shown in Case Study 1. Here, `P` is only one unnamed predicate appearing in the head of a rule, which states that there are two researchers (A and B) and both authored a paper (C); P could thus be renamed as "coauthors".

Case Study 1 (coauthors)**.** Rules of the coauthors case study.

```
P(A,B) :- author(A,C), author(B,C), researcher(A),
researcher(B), paper(C). # P = coauthors
```

Unnamed predicates appearing both in the head and in the body The second case study, *family*, is described in Case Study 2 and contains several rules representing various family relationships. Unnamed predicates appear both in the head and in the body of the rules, with `h2` being the only exception, as it appears only on the head on a single rule.

Case Study 2 (family)**.** Rules of the family case study.

```
h0(X,Y) :- mother(X,Y). # h0 = parent
h0(X,Y) :- father(X,Y).
h1(X,Y) :- h0(X,Z), h0(Z,Y). # h1 = grandparent
ancestor(X,Y) :- h0(X,Y).
ancestor(X,Y) :- h0(X, Z), ancestor(Z,Y).
h2(X, Y, Z) :- ancestor(Z, X), ancestor(Z,Y). # h2 =
   common_ancestor
h3(X,Y) :- sister(X,Y). # h3 = siblings
h3(X,Y) :- sister(Y, X).
h3(X,Y) :- brother(X,Y).
h3(X,Y) :- brother(Y,X).
h4(X,Y) :- h0(PX, X), h0(PY,Y), h3(PX, PY), dif(PX,PY). #
   h4 = cousins
```

The third case study, *math*, shown in Case Study 3, is a set of rules representing basic mathematical operations. Also in this case, unnamed predicates can appear both in the head and in the body of the rules.

Case Study 3 (math)**.** Rules of the math case study.

```
A(X) :- integer(X), !. # isNumber
A(X) :- float(X).

P(X,Y) :- A(X), A(Y), X > Y. # greaterThan
Q(X,Y) :- A(X), A(Y), X >=Y. # greaterOrEqual
R(X,Y) :- A(X), A(Y), X < Y. # lessThan
S(X,Y) :- A(X), A(Y), X <=Y. # lessOrEqual
T(X,Y) :- A(X), A(Y), X = Y. # equalTo

B(X) :- A(X), 0 is mod(X,2). # isEven
C(X) :- A(X), not B(X).      # isOdd

D(X,Y) :- ( A(X), R(X,0) ->  # abs
   Y is -X
   ;
   Y is X
).

E(X,Y,Z) :- A(X), A(Y), Z is X + Y. # sum
F(X,Y,Z) :- A(X), A(Y), Z is X - Y. # difference
G(X,Y,Z) :- A(X), A(Y), Z is X * Y. # product
H(X,Y,Z) :- A(X), A(Y), dif(Y,0), Z is X / Y. # quotient

L(X,Y) :- A(X), A(Y), dif(X,0), 0 is mod(Y,X). # isDivisor

M(0,Y,Y):- A(Y), P(Y,0). # gcd
M(X,0,X):- A(X), P(X,0).
M(X,Y,Z):- A(X), A(Y), P(X,0), P(Y,0), R(X,Y), M(Y,X,Z).
M(X,Y,Z):- A(X), A(Y), P(X,0), P(Y,0), P(X,Y), T is X mod Y, M(Y,T,Z).

N(X,0,0). # lcm
N(0,X,0).
N(X,Y,Z) :-
   A(X), A(Y),
   D(Y,U), D(X,K),
   M(X,Y,W),
   H(K,W,V),
   G(U,V,Z).
```

Unnamed predicates appearing only in the body We also created simpler one-rule examples to investigate the ability of the LLMs to find a meaningful name for predicates appearing only in the body. These examples are taken from the previous

ones and are shown in Case Study 4: the *grandparent* rule has predicate `h0` in the body, representing the `parent` relation; the *cousins* rule with predicate `h3` representing the `siblings` relation; and lastly, the *lcm* rule, representing the computation of the least common multiple using the greatest common divisor, uses predicates `G` and `H`, being `product` and `quotient`, respectively. In all these cases, the unnamed predicates appear only in the body of the rule.

Case Study 4. Rules of the *grandparent*, *cousins*, and *lcm* case studies.

```
grandparent(X,Y) :- h0(X,Z), h0(Z,Y). # h0 = parent
-----------------------------------------------------
cousins(X,Y) :- parent(PX, X), parent(PY,Y), h3(PX, PY),
                dif(PX,PY). # h3 = siblings
-----------------------------------------------------
least_common_multiple(X,Y,Z) :-
   G(X,Y,V), # G = product/multiply
   greatest_common_divisor(X,Y,W),
   H(V,W,Z). # H = quotient/divide
```

Unnamed predicates in a real-world dataset In addition, we also tried a real-world knowledge base taken from the mutagenesis dataset [158], a classic ILP benchmark dataset for Quantitative Structure-Activity Relationship (QSAR), that is, predicting the biological activity of chemicals from their physicochemical properties or molecular structure. The goal is to predict the mutagenicity of compounds given their chemical structure. We used the *ring_theory* provided with the dataset and removed the names of most of the head predicates. The set of rules is extensive, with 24 head predicates to be renamed, such as those in Case Study 5.

Case Study 5 (Mutagenesis)**.** Examples of rules of the Mutagenesis ring theory. Here, the "@>" operator checks if the first term is greater than the second term according to Prolog's standard order of terms.

```
anthracene(Drug,[Ring1,Ring2,Ring3]) :-
   benzene(Drug,Ring1),
   benzene(Drug,Ring2),
   Ring1 @> Ring2,
```

```
    interjoin(Ring1,Ring2,Join1),
    benzene(Drug,Ring3),
    Ring1 @> Ring3,
    Ring2 @> Ring3,
    interjoin(Ring2,Ring3,Join2),
    \+ interjoin(Join1,Join2,_),
    \+ members_bonded(Drug,Join1,Join2).

no_of_nitros(Drug,No) :-
    setof(Nitro,nitro(Drug,Nitro),List),
    length(List,No).

ring6(Drug,[Atom1|List],[Atom1,Atom2,Atom4,Atom6,Atom5,
    Atom3],
    [Type1,Type2,Type3,Type4,Type5,Type6]) :-
    bondd(Drug,Atom1,Atom2,Type1),
    memberchk(Atom2,[Atom1|List]),
    bondd(Drug,Atom1,Atom3,Type2),
    memberchk(Atom3,[Atom1|List]),
    Atom3 @> Atom2,
    bondd(Drug,Atom2,Atom4,Type3),
    Atom4 \== Atom1,
    memberchk(Atom4,[Atom1|List]),
    bondd(Drug,Atom3,Atom5,Type4),
    Atom5 \== Atom1,
    memberchk(Atom5,[Atom1|List]),
    bondd(Drug,Atom4,Atom6,Type5),
    Atom6 \== Atom2,
    memberchk(Atom6,[Atom1|List]),
    bondd(Drug,Atom5,Atom6,Type6),
    Atom6 \== Atom3.
```

Some of these rules are relatively short (1 to 3 body atoms), while others are much more complex, with more than 5 body atoms.

Unnamed predicates in an actual POPPER output To test our approach on an actual ILP theory, we designed a new case study and used POPPER to learn a recursive definition of a reachability predicate over a graph defined by bidirectional (road/2) and unidirectional (one_way/2) connections. The goal was to induce a definition of reachable/2 using predicate invention: POPPER introduces an intermediate invented predicate (e.g., inv1/2) to generalize a single-step path relation. The resulting program is presented in Case Study 6. Given that this case study was specifically designed by the authors for this work, it is not present in the existing literature to the best of our knowledge. As a result, it is unlikely that the LLMs were exposed to it during training.

Case Study 6 (reachability)**.** Rules of the reachability case study.

```
inv1(V0,V1):- road(V0,V1).
inv1(V0,V1):- one_way(V0,V1).
reachable(V0,V1):- inv1(V0,V1).
reachable(V0,V1):- reachable(V2,V1),inv1(V0,V2).
```

13.2.2 Prompts

We use zero-shot prompting for all the examples. Figure 13.2 shows *Prompt 1*, used to ask for name suggestions. We insert the corresponding rules with placeholders where `[rules]` is present. The prompt is built in sections, separated by `### section ###`: in the beninning, we provide the context; then, we give the instruction; lastly, we provide the desired output format. The prompts used for the examples differ only in the predicates listed in the instruction and output format. Since we don't want our method to be dependent on the chosen language, we instruct the LLMs to behave as a "specialist of logic programming".

Prompt 2, shown in Figure 13.3, is used when asking the models to choose between their own suggestions. For each model, `[suggested names]` is replaced with the list of its own suggestions. Since models can suggest names with semantically equivalent meanings but written in different formats, such as using underscores, hyphens, or capitalized letters, we ensure that they are standardized: we remove any underscores or hyphens from the name and convert them to camel case, leaving the first letter in lowercase. After this, we remove duplicates from the list.

```
### Context ###
You are a software engineer specialized in logic
   programming.

### Instruction ###
Given these first-order logic rules:
[rules]

Assign general and meaningful names to predicates h0, h1,
   h2, h3, and h4.
Do NOT change the body and the variables of the rules.
Give only ONE suggestion for each predicate.

### Output format ###
h0:
h1:
h2:
h3:
h4:
```

Fig. 13.2: *Prompt 1* for asking for one suggestion for each unnamed predicate. This prompt was used for Case Study 2. `[rules]` is replaced with the corresponding rules.

Finally, *Prompt 3* in Figure 13.4 was used for the judgment, where we asked the model to score the suggestions for each predicate. We also provide the scoring rules in the `INSTRUCTIONS` section.

13.2.3 LLMs used in this study

For this study, we selected 7 different LLMs: OpenAI's ChatGPT based on GPT-4o (ChatGPT-4o) [183, 184] and ChatGPT based on o3mini (ChatGPT-o3mini) [35]; Meta's Llama 3.2 3B Instruct (Llama) [185, 186], Google DeepMind's Gemini 1.5 Flash (Gemini) [187]; FalconMamba 7B Instruct (FalconMamba) [188] and Falcon3 7B Instruct (Falcon3) [189] of the Technology Innovation Institute (TII); and Cohere's Command R+ 08 2024 (Command R+) [190].

ChatGPT is a powerful chatbot running on GPT-4o (with approximately 200 billion parameters), capable of processing multi-modal inputs and answering within a few milliseconds, mirroring the behavior of a human. Generative pre-trained Trans-

```
### CONTEXT ###
You are a software engineer specialized in logic
   programming.
Given these first-order logic rules:
[rules]

The following names have been proposed for predicates h0,
   h1, h2, h3, and h4.
h0: [suggested names]
h1: [suggested names]
h2: [suggested names]
h3: [suggested names]
h4: [suggested names]

### INSTRUCTION ###
Choose the most suitable name for predicates h0, h1, h2,
h3, and h4, among the proposed ones.

### OUTPUT FORMAT ###
h0: CHOSEN_NAME
h1: CHOSEN_NAME
h2: CHOSEN_NAME
h3: CHOSEN_NAME
h4: CHOSEN_NAME
```

Fig. 13.3: *Prompt 2* used for asking the models to choose one among their own suggestions. This prompt was used for Case Study 2. `[rules]` is replaced with the corresponding rules and the suggestions of the model being used replace `[suggested names]`.

former (GPT) models, based on the tranformers technology, are robust and versatile and are nowadays widely employed for routine tasks such as text summarization or translation. Recently, given the significant advances, the adoption of ChatGPT and LLMs in general is also spreading in more critical applications [191], such as generating simplified clinical reports [192] or supporting decision-making process [193], with high accuracy and correctness. In addition to GPT models, OpenAI provides also reasoning models, trained with reinforcement learning. Reasoning models, like o1 and the latest o3mini that has 3B parameters, use a chain-of-thought approach, where intermediate reasoning steps are generated before producing the final answer. Although

this increases response time, reasoning models yield higher-quality outputs and are particularly effective in tasks involving science, mathematics, and coding [35]. Google DeepMind's Gemini 1.5 Flash is a fast and versatile model that can take text, images, audio, and video as input, and produces a text output. Gemini 1.5 Flash has been adopted in an AI-based application to help users monitor and reduce their carbon emissions [194]. It also obtained good results in answering multiple-choice questions regarding pediatric nephrology [195]. Llama 3.2 3B Instruct is a lightweight (3.21B parameters) text-only model of the Llama 3.2 family, which is a collection of multilingual, pre-trained, and instruction-tuned LLMs. FalconMamba-7B-Instruction is a version of the base model FalconMamba 7B, fine-tuned on instruction data. The model is based on Mamba [196], an attention-free architecture, and has 7.27B parameters. The Falcon3 family is a collection of pre-trained and instruction-tuned, decoder-only LLMs of various sizes. Falcon3-7B-Instruct, specifically, has been pre-trained on curated high-quality and multilingual data from various sources (web, code, STEM), and according to TII it achieves state-of-art-results on different tasks, including natural language processing, common-sense reasoning, coding, and reasoning. This is due to the improvements made in the development of the base models, such as depth upscaling and knowledge distillation [197]. Cohere's Command R+ 08-2024 is a multilingual model optimized for various applications, such as text summarization, reasoning, and structured data analysis, with 104 billion parameters, and trained on an extensive corpus of data in different languages. Command R+ 08-2024 has shown good performances for most of the writing tasks in Arabic tested by [198] and evaluated on *Gazelle*, the dataset they proposed, delivering clear and coherent responses.

The experiments involving incremental prompting, the *reachability* case study, and the human judgment evaluation were conducted between July and August 2025, while all other experiments took place between January and February 2025. Since ChatGPT-4o and ChatGPT-o3mini were no longer available at the time of the *reachability* case study experiment, we used ChatGPT-5, the only freely accessible version at that time.

13.2.4 Evaluation: LLM-as-a-Judge and human judgment

We use LLMs also for the automated evaluation of the answers, adopting the LLM-as-a-Judge strategy [21, 199].

We selected four models, ChatGPT-4o, ChatGPT-o3mini, Gemini, and Command

R+, to act as judges. Each judge assigns a score to every proposed name for each predicate: 1 for a correct and precise name, 0.5 for a name that is imprecise or too general but still correct, and 0 for a wrong name. The scores are then summed to determine the final results, and the name with the highest score is selected.

Finally, to assess the ability of LLMs in both renaming and judgment, we conducted a complementary experiment involving 14 human judges on all the case studies except Mutagenesis and *reachability*. The *reachability* case study was not included in the human judgment experiment, as it was introduced after the experiment had already been conducted, while Mutagenesis requires a field expert to provide reliable feedback. Participants were asked to score the proposed names using the same evaluation method adopted by LLMs judges.

13.3 Experiments and Results

We used the web interface to execute both ChatGPT running on GPT-4o and ChatGPT running on GPT-o3mini. We accessed Gemini and Command R+ via their own API. For Llama, FalconMamba, and Falcon3, experiments were performed locally via the Hugging Face Transformers framework [200] on a cluster consisting of two machines with two AMD EPYC 9654 96-cores and four machines with two AMD EPYC 9454 48-cores. The memory limit was set to 100 GB, and the time limit was set to 17 hours and 30 minutes.

13.3.1 Prompting strategy

We carried out several initial prompt design attempts to observe how LLMs responded under varying instructions. For instance, after noticing that the models often provided multiple equivalent suggestions, we decided to limit them to just one, instructing them to "`Give only ONE suggestion for each predicate`". We also added a constraint to prevent models from altering the structure of the rules, after observing such behaviors with some LLMs. Each of these attempts was executed independently, ensuring that no contextual information was shared between runs. Importantly, the models did not retain any memory of previous executions; in particular, ChatGPT was used with the memory feature explicitly disabled to prevent any persistence of prior interactions. After analyzing the models' responses and identifying recurring issues or undesired be-

haviors, we refined our instructions iteratively. The final version of the prompt, shown in Figure 13.2, was then used to rerun all experiments from scratch. Additionally, we observed that models sometimes gave the correct response on the first try, while other times they required several attempts. To address this, and given the non-deterministic nature of the answers, we decided to repeat the question a certain number of times to address this issue. In these preliminary experiments, we decided to use three repetitions. All the models handled the task appropriately, so zero-shot prompting was sufficient in this case. Most of them followed the instructions accurately, with only a few exceptions. For example, Llama usually wrote a sentence (explaining the reasons for its choices) instead of adopting the requested format. In such cases, however, the name can be extracted from the answer using basic string manipulation. After getting three suggestions out of each model, we asked them to choose only one name among their own ones.

Tables 13.1 and 13.2 show the final choices made by each model for each predicate of each case study; we also report for each LLM how many of the suggested names could be considered correct. Table 13.3 provides a summary of the number of correct name suggestions per predicate. Each model receives a score of 1 if its final suggestion matches the expected name, and 0 otherwise. In the final column, we indicate how many LLMs correctly named each predicate. This overview allows us to easily compare the performance of the models on a per-predicate basis and identify which predicates were more challenging.

Results for unnamed predicates appearing only in the head The *coauthors* case study, which was the simplest, presented no difficulties for the models. In fact, all of them suggested a correct name, although sometimes one that was too verbose.

Results for unnamed predicates appearing both in the head and in the body Regarding the *family* case study, the GPT models and Command R+ provided the correct name of each predicate, and Gemini failed to identify only h4. Llama was completely unable to find meaningful names for most of the predicates, with only two correct suggestions. Furthermore, one out of the three times it rewrote most of the rules, making its answer completely wrong. FalconMamba and Falcon3 also struggled with the *family* case study, as none of the answers were correct. In particular, Falcon3 provided the answer for only one predicate, likely being misled by the word “ONE” in

the prompt. For this reason, its answers were not considered.

For the *math* case study, the GPT models correctly identified almost all the names. Gemini followed closely, with only two mistakes. Falcon3 and Command R+ also identified most of the correct names, while Llama just over half of them. On the other hand, FalconMamba simply rewrote the rules with different capital letters all three times, and therefore its answers were ignored. Many models couldn't find the correct name for predicates L (`isDivisor`, suggested only by Command R+), M (`gcd`), and N (`lcm`); these last two were correctly identified only by ChatGPT-4o and ChatGPT-o3mini.

Results for unnamed predicates appearing only in the body In the *lcm* case study, most models struggled to identify the correct names for predicates G and H, namely `product` and `quotient` (or equivalent terms). Only ChatGPT-4o and ChatGPT-o3mini provided correct suggestions. Finding `siblings` for predicate h3 in the *cousins* case study was also challenging for some models, even if they suggested it in the *family* case study. Only the GPT models, Gemini and LLama provided the correct name. For the *grandparent* case study, all the models correctly suggested `parent` for predicate h0, except for FalconMamba.

Results for the Mutagenesis case study In Table 13.4 we report the results obtained on the Mutagenesis ring theory, using ChatGPT-4o and ChatGPT-o3mini. Some of the predicates appear to be similar to the original ones, but overall it is difficult to determine whether the suggested names are meaningful. In such a case, a domain expert should make the final decision. To improve the quality of the output, the models could be fine-tuned on the specific domain; alternatively, context (e.g., in the form of scientific papers) could be provided.

It is worth mentioning another attempt we made with Mutagenesis. For this initial experiment, we used a less constrained version of *Prompt 1*, where we simply asked for suggestions for all the predicates, without specifying the context or limiting the model to provide only one suggestion per predicate. The response of Command R+ was particularly interesting. The model not only explained the predicate to rename but in some cases it also invented a helper predicate for it. An example of such behavior is shown in Figure 13.5: the model created the helper predicate `check_atom_count_recursive` for predicate HP19, which was renamed as `check_atom_count_recursive`. Regardless

of the correctness of the suggested name and despite altering the structure of the rule, this suggests the potential of LLMs to directly perform PI.

Results for the reachability case study The *reachability* case study was generally unchallenging for the majority of models evaluated. Suggestions produced by GPT-5 (direct_connection), Command R+ (is_connected), Gemini (directly_connected), and LLama (is_connected) are all plausible and appropriate, as they generalize a single-step path relation. In contrast, both FalconMamba and Falcon3 exhibited notable difficulties. FalconMamba generated an output (can reach) that was syntactically invalid as a predicate, due to the inclusion of a space in the identifier, while Falcon 3 failed to produce any answer at all.

13.3.2 Results for the incremental prompting experiment

We additionally conducted an incremental prompting experiment with the LLMs that demonstrated lower performance, namely FalconMamba, Falcon3, and Llama, with the goal of improving their answers. We focused on the *math* case study, which was selected due to its potential to highlight the benefits of incremental prompting. In this experiment, suggestions were requested for one predicate at a time, while also providing the model with the correct names of previously addressed predicates. Figure 13.6 shows the prompts used in the incremental prompting experiments with answers from Falcon3. The initial prompt, *Prompt 0*, requests the name of only the first predicate (A). Then, its answer is used to replace the corresponding unnamed predicate in the following prompt, *Prompt 1*, while requesting a suggestion for the second predicate (P). This continues until the last prompt, in which all the unnamed predicates of the math case study have been renamed, except of the last one. This incremental approach aimed to support the model by narrowing its focus and reducing the complexity of the task, in contrast to requiring it to generate names for all predicates simultaneously. However, the outcomes did not meet expectations. None of the models showed improved performance in terms of correctly identifying a greater number of names; in some instances, performance even declined.

Table 13.5 shows the results of this experiment. With incremental prompting Falcon3 correctly identified 10 out of 16 predicate names, a decrease from the 12 out of 16 correctly identified with zero-shot prompting. Similarly, Llama achieved 8 correct

predictions with incremental prompting, compared to 9 without. A substantial improvement was observed with FalconMamba. With zero-shot prompting, the model failed to generate a meaningful output, simply repeating the predicates. In contrast, when incremental prompting was applied, the model provided a suggestion for all the predicates, finding 11 correct answers out of 16. These findings, particularly in the case of FalconMamba, indicate that breaking down the prompt and using incremental prompting can significantly enhance performance in models that otherwise struggle with zero-shot prompting.

13.3.3 LLMs judgment results

Regarding the judgment on the *coauthors* case study, shown in Table 13.6, all the suggestions are plausible. LLM judges seem to prefer a longer and more detailed name, assigning the lowest score to the simplest one, i.e. `coauthors`.

In Table 13.7 we report the results of the judgment on the *family* case study. The correct name was chosen for each predicate. The same holds for the *grandparent* and *cousins* case studies, both shown in Table 13.6.

Table 13.8 shows the results of the judgment on the *math* case study. Here, the correct name was chosen for all the predicates, except for predicate L. While the correct name is `divisor`, it was mostly interpreted as "divisible", which is actually the opposite. Regarding predicates M and N, all the judges recognized the true names, `gcd` and `lcm` respectively, even if they didn't suggest them initially. The strategy of asking multiple models for name suggestions turned out to be effective, as it allowed the judges to identify the correct name when one model suggested it, even when the others did not. The same cannot be said for the *lcm* case study, as can be seen in Table 13.6. For predicate G, `findLeastCommonMultipleIntermediate` was chosen as the most fitting name, instead of `computeProduct` or `multiply`, which would have been equally correct. Similarly, for predicate H, `computeLcmFromGcd` was selected as the most appropriate name, rather than `divide` or `divideValues`.

Table 13.9 presents the judgment results for the reachability case study. It is important to note that ChatGPT-5 was used as a substitute for ChatGPT-4o and ChatGPT-o3mini, as these models were no longer available at the time the experiment was conducted. Three out of the four alternatives were selected as the most suitable name, each receiving the same score. This outcome is acceptable, as all three options

are considered plausible. Notably, however, only ChatGPT-5 assigned a score of 0 to the proposed name `can reach`, which is syntactically incorrect. In contrast, both Gemini and Command R+ assigned it a score of 0.5, indicating a limited ability to recognize the syntactic invalidity of the suggestion.

13.3.4 Human judgment results

In this experiment, we involved students from the master's course in Artificial Intelligence of the University of Ferrara who had previously completed a course on logic programming. These students were asked to evaluate the predicate names assigned by LLMs. They completed a form containing the same prompt and the proposed names that were given to the LLMs judges. Out of the 56 students we contacted, 14 responded. Although all the proposed alternatives in the *coauthors* case study are plausible, human judges did not select the most straightforward option for predicate P, that is, `coauthors`. This behavior aligns with that observed with the LLMs. They also correctly identified the appropriate names for predicates h0 and h3 in the *grandparent* and *cousins* case studies, respectively. In the *lcm* case study, human judges successfully selected the correct names for predicates G and H, whereas the LLMs judges failed to do so. Averaged scores assigned by human judges for these case studies are summarized in Table 13.10.

Table 13.11 presents the results of the human evaluation for the proposed names in the *family* case study. For each predicate, the correct name received the highest overall score. However, it is noteworthy that several human judges assigned non-zero scores to incorrect names. For instance, in the case of predicate h0, 4 out of 14 judges assigned a score of 1, and another 4 assigned a score of 0.5 to the incorrect name `ancestor`.

For the *math* case study, human judges consistently assigned the highest score to the correct name across all predicates, as shown in Table 13.12. Moreover, they successfully identified the correct name for predicate L, that is `isDivisor`, which none of the LLMs judges were able to recognize correctly.

13.3.5 Summary of results

Overall, ChatGPT-4o and ChatGPT-o3mini appear to be the best-performing models, followed by Gemini and Command R+; Llama, FalconMamba, and Falcon3 performed the weakest.

It is worth noting that we chose some relatively small models to test whether they could identify useful names. Llama 3.2 and Falcon3 provided several correct suggestions, though not for every predicate. While incremental prompting did not enhance performance for Falcon3 and Llama, it led to a substantial improvement for Falcon-Mamba. Nevertheless, their smaller size allows them to be run locally on machines with lower processing power, making this approach accessible to users without high-performance hardware.

The evaluation strategy that we adopted is automatic, meaning that human intervention is not required except for a final assessment, which should be faster than evaluating every single suggestion for each predicate. It could be argued that an LLM could make errors not only in making suggestions but also in judging them. Even though in these experiments we asked the LLMs to score only once, for a more robust evaluation, the procedure could be repeated multiple times per judge, and the results averaged. If a tie occurs between two or more suggestions, a new evaluation could be carried out, or the final decision could be left to domain experts.

Based on the results of the human evaluation, we can conclude that, overall, LLMs were generally capable of performing both the renaming and judgment tasks effectively. Their assessments aligned with those of human judges in most cases, with discrepancies emerging primarily in the more challenging or ambiguous predicates.

13.3.6 Limitations

As this represents an initial exploration, there are some limitations that need to be discussed. The main one is the use of relatively simple examples. The rules express common knowledge and typical relationships, which the models may have already seen in the LP codes they have been trained on. Nonetheless, this initial step was necessary to assess whether LLMs could correctly process them; otherwise, their ability to grasp even more specific relationships would have seemed unlikely. Despite the simplicity of the examples, we encountered several challenges throughout the process. One issue is that, at times, some models fail to follow the request or the output format, which complicates the automation of the entire process. For instance, in some preliminary experiments, we noticed that some models also altered the body of the rule, even if instructed to find a name for the head predicate. This is not desirable, as it could completely change the meaning of the logic theory, which is assumed to be correct. That

is why we decided to add "Do NOT change the body of the rules" to the instructions. Another constraint, "Give only ONE suggestion for each predicate.", was added to the prompt to limit the number of suggestions. While one could argue that this might cause the potentially correct answer to be overlooked, we addressed this concern by repeating the request multiple times. When this behavior persisted, we decided to disregard that particular answer for that specific predicate. Even if one answer is discarded, the presence of multiple models offering suggestions reduces the impact of this issue. Models also struggled to recognize that some names, despite being written in different formats, were conceptually equivalent (e.g., *lessThan*, *less_than*, *LessThan*), even when explicitly instructed to treat them as equal. As a result, it was necessary to manually standardize these names to ensure that they were evaluated as the same. The issue persists to some extent: singular and plural forms are considered different (e.g., *cousins* and *cousin*), as well as names that include a verb or particle (e.g., *isEqual* or *equalTo*). Lastly, this approach was unsuccessful when tested on predicates from a real-world dataset, Mutagenesis. It should be noted that nearly all of the predicates' true names were masked, which significantly hindered the models' ability to identify them and grasp the relationship between them. However, such a scenario is highly unlikely, if not entirely unrealistic, since it would mean that the background knowledge is almost empty. In this case, inferring a set of meaningful rules would be highly difficult for any ML model. Nevertheless, our findings hint that domain-specific fine-tuning might be necessary for real-world applications [201], provided that enough computational resources are available. Alternatively, big models could be replaced by smaller or quantized fine-tuned versions, which are more computationally efficient and have been shown not to significantly decrease the quality of the results [19, 202, 203].

Despite these limitations, the results obtained from some hand-crafted sets of rules prove that LLMs are a promising solution to find suggestions for unnamed predicates in logic theories.

13.4 Related work

PI is considered a critical step, yet it is still not widely spread [65]. Choosing suitable names for newly created predicates is an even more neglected area in ILP and has not received much attention so far, with only a few attempts made in the past.

In [204] the authors adopted a meta-level abduction approach to identify previously

unknown relations within a knowledge graph that represents a specific domain. In this setting, new predicates are invented as existentially quantified variables, which act as placeholders for the missing or hidden knowledge. While the system is capable of structurally generating these new predicates, their semantic meaning (i.e., their names) is not determined automatically and must instead be manually defined or interpreted based on the domain context. This highlights a key limitation: the invented predicates are syntactically valid but lack meaningful, human-readable names without further domain-specific interpretation. In the rule abduction approach proposed in [205], the authors built databases related to the cello playing domain to assist performers in assigning meaningful names to new abduced rules.

As the power of LLMs continues to grow, we could leverage their capabilities to enable them to reason about rules and generate meaningful names for previously unnamed predicates. While many have highlighted the impressive performances of LLMs with zero or few-shot prompting [24, 33, 34], a rising number of recent works have explored their reasoning ability on more complex tasks [206] involving common knowledge and planning [207, 208], math [34, 209], and symbolic reasoning [34, 210, 211]. In particular, LLMs are already widely employed in software engineering [24, 25, 212, 213], where they typically generate code with semantically meaningful variable names.

[206] evaluated state-of-the-art LLMs on an ILP benchmark and found their performance lacking when compared to a smaller neural program induction model. They evaluated LLMs on two benchmarks, one of which required the model to induce programs to deduce more complex family relationships from some basic ones. Although this setup is similar to ours, they focused on assessing the relational reasoning capabilities of LLMs using standard natural language and truth value prompting. In contrast, our goal was to determine if LLMs could identify names that are relevant to the context of the example. [214] used GPT-3.5 and GPT-4 for property induction, a traditional inductive reasoning task that involves generalizing a property from a few examples to new cases. In their experiments, participants—including both humans and language models—were presented with a set of objects sharing a property and asked to infer whether new objects shared the same property. Although GPT-3.5 had difficulty capturing various aspects of human behavior, GPT-4's performance was mostly close to that of humans.

[215] investigated the ability of pretrained language models to infer natural language rules from natural language facts, obtaining promising results. [216] proposed

an approach to improve LLMs' inductive reasoning ability. They generate hypotheses in natural language, and then translate them into specific programs used for making predictions, breaking the task into two levels of abstraction, and outperforming current baselines. In [217], the authors explored techniques to extract and leverage the knowledge embedded in LLMs by converting it into different types of logic programs, aiming to improve reasoning capabilities and ensure that LLM outputs are consistent with their intended purposes. Additionally, [218] aimed to determine whether LLMs can address ILP tasks, analyzing their capabilities and limitations on logic theory induction. Their results indicate that while larger LLMs can achieve competitive results compared to state-of-the-art ILP systems, long predicate relationship chains are still an obstacle for LLMs in inductive reasoning tasks. [219] proposed NeSyGPT, a novel architecture that integrates neural and symbolic reasoning, that fine-tunes a vision-language foundation model to extract symbolic features from raw data, and then learns an expressive answer set program using them. The results show that NeSyGPT outperforms various baselines in accuracy and can be used for more complex NeSy tasks. They thus demonstrated that LLMs can reduce the need for manual efforts in automating the interface between neural and symbolic components. Remaining in the NeSy field, [220] presented a novel approach to expand the knowledge base by integrating LLMs with structured semantic representations. It employs a state-of-the-art LLM to produce a natural language description of an input image, which is then transformed into an Abstract Meaning Representation (AMR) graph. This graph is further extended with logical design patterns and layered semantics derived from linguistic and factual knowledge bases. [221] proposed LLM2LAS, a system that learns commonsense knowledge in the form of ASP programs, from story-based question and answers in natural language, using only a few examples and combining LLMs' semantic parsing abilities with ILASP [222], a tool for learning ASP programs. The authors of [210] used an LLM to generate ASP programs representing logic puzzles, given their description in natural language. Here, the authors state that the LLM successfully produced an ASP, despite some minor errors that were corrected manually, confirming the possibility of leveraging LLMs to support the creation of ASP programs. Although some findings suggest that LLMs still struggle with complex reasoning [206, 223], this proves that the field yields some potential and is worth further investigation, especially considering the continuous improvements of LLMs.

```
### CONTEXT ###
You are a software engineer specialized in logic
   programming.
The following logic rules have unnamed predicates in the
   head and/or in the body:
[rules]

These are the suggested names for the unnamed predicates h0
   , h1, h2, h3, and h4:
h0: [suggested names]
h1: [suggested names]
h2: [suggested names]
h3: [suggested names]
h4: [suggested names]

### INSTRUCTIONS ###
Score the proposed names for each predicate following the
   rules below:
- Assign 1 to correct and precise answers.
- Assign 0.5 to answers that are too generic, but still
   correct.
- Assign 0 to incomplete, imprecise or incorrect answers.

### OUTPUT FORMAT ###
For each predicate, list all the suggestions and their
   score:
h0:
h1:
h2:
h3:
h4:
```

Fig. 13.4: *Prompt 3* used for asking the models acting as judges to score the suggestions for each predicate. This prompt was used for Case Study 2. `[rules]` is replaced with the corresponding rules.

Predicate	ChatGPT-4o	ChatGPT-o3mini	Command R+
Coauthors[1]			
P	coauthors_with_ common_paper	coauthors	co_authored_paper
correct	**1/1**	**1/1**	**1/1**
Family			
h0	parent	parent	parent
h1	grandparent	grandparent	grandparent
h2	common_ancestor	common_ancestor	common_ancestor
h3	sibling	sibling	sibling
h4	cousin	cousin	cousins
correct	**5/5**	**5/5**	**5/5**
Grandparent			
h0	parent	parent	parent
correct	**1/1**	**1/1**	**1/1**
Cousins			
h3	siblings	siblings	different_parents
correct	**1/1**	**1/1**	**0/1**
Math			
A	is_number	numeric_value/1	is_integer
B	is_even	even/1	is_even
C	is_odd	odd/1	is_odd
D	absolute_value	abs_value/2	negate_if_negative
E	add	sum/3	sum
F	subtract	difference/3	difference
G	multiply	product/3	product
H	divide	quotient/3	division
L	is_divisible	divides/2	is_divisor
M	gcd	gcd/3	modular_arithmetic
N	lcm	lcm/3	nested_calculation
P	greater_than	greater_than/2	greater_than
Q	greater_than_or_equal	greater_or_equal/2	greater_equal
R	less_than	less_than/2	less_than
S	less_than_or_equal	less_or_equal/2	less_equal
T	equal	equal_to/2	equal
correct	**15/16**	**15/16**	**12/16**
lcm			
G	compute_product	multiply	is_multiple_of
H	divide_values	divide	lcm_calculation
correct	**2/2**	**2/2**	**0/2**
reachability			
inv1	direct_connection[2]	*NA*	is_connected
correct	**1/1**	**-**	**1/1**

Table 13.1: Final names suggestions for each predicate of all the examples, made by ChatGPT-4o, ChatGPT-o3mini, and Command R+.

[1] In this case, several names were suggested and even if excessively verbose, they can be considered correct.

[2] Since ChatGPT-4o and ChatGPT-o3mini were no longer available at the time of the experiment, ChatGPT-5 was used instead.

Predicate	FalconMamba	Falcon3	Gemini	Llama
Coauthors[1]				
P	Co-authoredResearchPaper	CoAuthorResearchers	coauthored_ research_paper	authoredTogether
correct	**1/1**	**1/1**	**1/1**	**1/1**
Family				
h0	ancestor	-	parent	parent
h1	ancestor	-	grandparent	ancestor
h2	ancestor	-	commonAncestor	great_ancestor
h3	sister	-	sibling	sibling
h4	h3	-	halfSibling	full_sibling
correct	**0/5**	**0/5**	**4/5**	**2/5**
Grandparent				
h0	rename_h0	parent	parent	parent
correct	**0/1**	**1/1**	**1/1**	**1/1**
Cousins				
h3	is_third_degree_relative	parent	sibling	sibling
correct	**0/1**	**0/1**	**1/1**	**1/1**
Math				
A	-	IsNumber	isNumber	is_integer
B	-	IsEven	isEven	is_odd
C	-	IsOdd	isOdd	is_even
D	-	IsPositiveDifference	absValue	next
E	-	Sum	sum	add
F	-	Difference	difference	subtract
G	-	Product	product	multiply
H	-	Division	quotient	divide
L	-	DivisibleBy	isDivisibleBy	modulo
M	-	CustomOperation	gcd	accumulate
N	-	ComplexOperation	complexCalculation	accumulate_sequence
P	-	GreaterThan	greaterThan	greater_than
Q	-	GreaterThanOrEqual	greaterThanOrEqual	greater_than or_equal_to
R	-	LessThan	lessThan	less_than
S	-	LessThanOrEqual	lessThanOrEqual	less_than_ or_equal_to
T	-	Equal	isEqual	equal_to
correct	**0/16**	**12/16**	**14/16**	**9/16**
lcm				
G	lowest_common_multiple	multiply_and_add	find_least_common_ multiple_intermediate	greatest_ common_divisor
H	highest_common_divisor	divide_and_subtract	compute_lcm_ from_gcd	least_ common_multiple
correct	**0/2**	**0/2**	**0/2**	**0/2**
reachability				
inv1	can reach	-	directly_connected	is_connected
correct	**0/1**	**0/1**	**1/1**	**1/1**

Table 13.2: Final names suggestions for each predicate of all the examples, made by FalconMamba, Falcon3, Gemini, Llama.

[1] In this case, several names were suggested and even if excessively verbose, they can be considered correct.

Predicate	ChatGPT 4o	ChatGPT o3mini	Command R+	Falcon Mamba	Falcon3	Gemini	Llama	**Total**
Coauthors								
P	1	1	1	1	1	1	1	**7/7**
Family								
h0	1	1	1	0	0	1	1	**5/7**
h1	1	1	1	0	0	1	0	**4/7**
h2	1	1	1	0	0	1	0	**4/7**
h3	1	1	1	0	0	1	1	**5/7**
h4	1	1	1	0	0	0	0	**3/7**
Grandparent								
h0	1	1	1	0	1	1	1	**6/7**
Cousins								
h3	1	1	0	0	0	1	1	**4/7**
Math								
A	1	1	0	0	1	1	0	**4/7**
B	1	1	1	0	1	1	0	**5/7**
C	1	1	1	0	1	1	0	**5/7**
D	1	1	0	0	0	1	0	**3/7**
E	1	1	1	0	1	1	1	**6/7**
F	1	1	1	0	1	1	1	**6/7**
G	1	1	1	0	1	1	1	**6/7**
H	1	1	1	0	1	1	1	**6/7**
L	0	0	1	0	0	0	0	**1/7**
M	1	1	0	0	0	1	0	**3/7**
N	1	1	0	0	0	0	0	**2/7**
P	1	1	1	0	1	1	1	**6/7**
Q	1	1	1	0	1	1	1	**6/7**
R	1	1	1	0	1	1	1	**6/7**
S	1	1	1	0	1	1	1	**6/7**
T	1	1	1	0	1	1	1	**6/7**
lcm								
G	1	1	0	0	0	0	0	**2/7**
H	1	1	0	0	0	0	0	**2/7**
reachability								
inv1	1[1]	-	1	0	0	1	1	**4/6**

Table 13.3: Summary of correct name suggestions for each predicate. For each LLM, a value of 1 indicates that its final suggestion was correct, while 0 indicates an incorrect suggestion. The *Total* column shows the number of LLMs that correctly named each predicate.

[1] Since ChatGPT-4o and ChatGPT-o3mini were no longer available at the time of the experiment, ChatGPT-5 was used instead.

```
% Rules for specific chemical patterns

% HP19: Checks if a drug contains a specific atom type with
    a certain number of occurrences.
% Args: Drug, Number of atoms, List of atoms, List of atom
   types
check_atom_count(Drug, N, Atoms, Types) :-
    atm(Drug, Atom, Type, _, _),
    Types = [Type | _],
    check_atom_count_recursive(Drug, N, [Atom], Atoms, Types
       , 1).

check_atom_count_recursive(_, 1, Atoms, Atoms, _, _) :- !.
   % Base case: N is 1
check_atom_count_recursive(Drug, N, [Atom | _], Atoms, [
   Type | Types], N) :-
    N > 1,
    N1 is N - 1,
    atm(Drug, Atom, Type, _, _),
    check_atom_count_recursive(Drug, N1, _, Atoms, Types, _)
       .
```

Fig. 13.5: Result obtained with Command R+ on Case Study 5 with a less constrained version of *Prompt 1*.

Predicate	ChatGPT-4o	ChatGPT-o3mini
HP1 (anthracene)	fused_aromatic_ tricycles_non_overlapping	fused_aromatic_tricyclic_ system_no_bridge
HP2 (phenanthrene)	fused_aromatic_ tricycles_adjacent	fused_aromatic_tricyclic_ system_with_bridge
HP3 (ball3)	fused_aromatic_ tricycles_interconnected	fused_aromatic_tricyclic_ system_common_junction
HP4 (members_bonded)	shared_aromatic_bond	ring_junction_bridge
HP5 (no_of_benzenes)	aromatic_hexagons_count	count_aromatic_ six_membered_rings
HP6 (no_of_carbon_5_ aromatic_rings)	aromatic_pentagons_count	count_aromatic_ five_membered_rings
HP7 (no_of_carbon_6_rings)	non_benzenoid_ hexagons_count	count_nonaromatic_ six_membered_rings
HP8 (no_of_carbon_5_rings)	non_benzenoid_ pentagons_count	count_nonaromatic_ five_membered_rings
HP9 (no_of_hetero_ aromatic_6_rings)	heterocyclic_hexagons _count	count_aromatic_heterocyclic _six_membered_rings
HP10 (no_of_hetero_ aromatic_5_rings)	heterocyclic_pentagons_count _count	count_aromatic_heterocyclic five_membered_rings
HP11 (ring_size_6)	hexagonal_ring_structure	generic_six_ membered_ring
HP12 (ring_size_5)	pentagonal_ring_structure	generic_five_membered_ring
HP13 (benzene)	benzene_ring	aromatic_six_membered_ring
HP14 (carbon_5_ aromatic_ring)	cyclopentadiene_ring	aromatic_five_ membered_ring
HP15 (carbon_6_ring)	non_benzenoid_hexagon	nonaromatic_six_ membered_ring
HP16 (carbon_5_ring)	non_benzenoid_pentagon	nonaromatic_ five_membered_ring
HP17 (hetero_ aromatic_6_ring)	heterocyclic_hexagon	aromatic_heterocyclic_ six_membered_ring
HP18 (hetero_ aromatic_5_ring)	heterocyclic_pentagon	aromatic_heterocyclic_ five_membered_ring
HP19 (atoms)	atom_sequence	ring_atom_chain
HP20 (ring6)	aromatic_hexagonal_ bond_pattern	six_membered_ ring_structure
HP21 (ring5)	aromatic_pentagonal_ bond_pattern	five_membered_ ring_structure
HP22 (no_of_nitros)	nitro_group_count	count_nitro_groups
HP23 (nitro)	nitro_group_structure	nitro_group_structure
HP24 (methyl)	methyl_group_structure	methyl_group_structure

Table 13.4: Results obtained on the Mutagenesis ring theory, with ChatGPT-4o and ChatGPT-o3mini.

Prompt 0:

```
### CONTEXT ###
You are a software engineer specialized in logic programming.

### INSTRUCTION ###
Given the logic rules below, find a meaningful name for A.
Give only ONE suggestion for the predicate name.
Write only the predicate name after the predicate placeholder, without any additional
    text.
Do not rewrite the predicate body.

A(X) :- integer(X), !.
A(X) :- float(X).

Strictly follow this output format:
A: [your_suggestion]
```

Response:

```
A: is_number
```

Prompt 1:

```
### CONTEXT ###
You are a software engineer specialized in logic programming.

### INSTRUCTION ###
Given the logic rules below, find a meaningful name for P.
Give only ONE suggestion for the predicate name.
Write only the predicate name after the predicate placeholder, without any additional
    text.
Do not rewrite the predicate body.

is_number(X) :- integer(X), !.
is_number(X) :- float(X).

P(X,Y) :- is_number(X), is_number(Y), X > Y.

Strictly follow this output format:
P: [your_suggestion]
```

Response:

```
P: is_greater_number
```

Fig. 13.6: Prompts used in the incremental prompting experiments with Falcon3 answers.

Predicate	FalconMamba	Falcon3	Llama
A : number	integer_or_float	is_number	has_integer_value
B : even	even_integer	is_even_number	parity
C : odd	odd_integer	is_odd_number	odd
D : abs	abs	negate_if_negative	divide_by_zero
E : addition	sum	add_numbers	sum
F : substraction	subtract	subtract_numbers	difference
G : multiplication	multiplication	multiply_numbers	multiply
H : division	division_by_non_zero_integer	divide_numbers	divide
L : divisor	even_integer	divisible_by	is_divisible
M : gdc	even_integer	swap_and_mod_numbers	min
N : lcm	even_integer_product	calculate_result	number
P : greater_than	greater_than_or_equal	is_greater_number	greater_integer
Q : greater_equal_than	greater_than_or_equal	is_number_or_equal	greater_than
R : less_than	less_than	is_less_number	less_than
S : less_equal_than	less_than	is_number_or_less	between
T : equal	equal_to	is_equal_number	equal
correct	**11/16**	**10/16**	**8/16**

Table 13.5: FalconMamba, Falcon3 and Llama suggestions for the predicates of the *math* case study, in the incremental prompting experiment.

Case Study - *Predicate*	ChatGPT (4o)	ChatGPT (o3mini)	Gemini	Command R+	**Score**
coauthors - *P*:					
coauthoredResearchPaper*	1	1	1	1	**1.000**
coauthorsWith-CommonPaper*	1	1	1	1	**1.000**
coAuthoredPaper*	0.5	1	0.5	1	**0.750**
coAuthorResearchers*	1	0.5	0.5	1	**0.750**
authoredTogether*	0.5	0.5	0.5	1	**0.625**
coauthors*	0.5	0.5	0.5	0.5	**0.500**
grandparent - *h0*:					
parent*	1	1	1	1	**1.000**
renameH0	0	0	0	0.5	**0.125**
cousins - *h3*:					
siblings*	1	1	1	0	**0.750**
sibling*	1	1	0	0	**0.500**
differentParents	0	0	1	1	**0.500**
isThirdDegreeRelative	0.5	0	0.5	0	**0.250**
parent	0	0	0	0	**0.000**
lcm - *G*					
findLeastCommon-MultipleIntermediate	1	0.5	1	0	**0.625**
computeProduct*	0.5	1	0.5	0	**0.500**
multiply*	0.5	1	0.5	0	**0.500**
greatestCommonDivisor	0	0	0	1	**0.250**
isMultipleOf	0	0	0	0	**0.000**
lowestCommonMultiple	0	0	0	0	**0.000**
multiplyAndAdd	0	0	0	0	**0.000**
lcm - *H*					
computeLcmFromGcd	1	1	1	0.5	**0.875**
divide*	0.5	1	0.5	0.5	**0.625**
lcmCalculation	1	0.5	0.5	0.5	**0.625**
divideValues*	0.5	1	0.5	0	**0.500**
leastCommonMultiple	0	0	0	1	**0.250**
highestCommonDivisor	0	0	0	0	**0.000**
divideAndSubtract	0	0	0	0	**0.000**

Table 13.6: LLM judgment results for Case Studies 1 (*coauthors*) and 4 (*grandparent*, *cousins* and *lcm*). Judges assign a score of 1 for correct and precise names, 0.5 for imprecise or overly general names that are still correct, and 0 for incorrect names. The final score, shown in the last column, is the average of the individual scores given by the judges. Names that can be considered correct are marked with an asterisk ("*").

Predicate	ChatGPT (4o)	ChatGPT (o3mini)	Gemini	Command R+	**Score**
h0					
parent*	1	1	1	1	**1.000**
ancestor	0.5	0	0.5	0.5	**0.375**
h1					
grandparent*	1	1	1	1	**1.000**
ancestor	0.5	0.5	0.5	0.5	**0.500**
h2					
commonAncestor*	1	1	1	0.5	**0.875**
ancestor	0.5	0.5	0.5	0	**0.375**
greatAncestor	0.5	0	0	0.5	**0.250**
h3					
sibling*	1	1	1	1	**1.000**
sister	0.5	0	0.5	0.5	**0.375**
h4					
cousin*	1	1	1	0.5	**0.875**
cousins*	1	1	0.5	0.5	**0.750**
halfSibling	0	0	0.5	0.5	**0.250**
fullSibling	0	0	0.5	0.5	**0.250**
h3	0	0	0	0	**0.000**

Table 13.7: LLM judgment results for Case Study 2 (*family*). Judges assign a score of 1 for correct and precise names, 0.5 for imprecise or overly general names that are still correct, and 0 for incorrect names. The final score, shown in the last column, is the average of the individual scores given by the judges. Names that can be considered correct are marked with an asterisk ("*").

Predicate	ChatGPT (4o)	ChatGPT (o3mini)	Gemini	Command R+	**Score**
A					
isNumber*	1	1	1	1	**1.000**
numericValue*	0.5	1	0.5	0.5	**0.625**
isInteger	0	0	0.5	0.5	**0.250**
B					
isEven*	1	1	1	1	**1.000**
even*	1	1	1	0.5	**0.875**
isOdd	0	0	0	0	**0.000**
C					
isOdd*	1	1	1	1	**1.000**
odd*	1	1	1	0.5	**0.875**
isEven	0	0	0	0	**0.000**
D					
absoluteValue*	1	1	1	1	**1.000**
absValue*	1	1	1	0.5	**0.875**
negateIfNegative	1	0.5	1	0.5	**0.750**
isPositiveDifference	0	0	0.5	0	**0.125**
next	0	0	0	0	**0.000**
E					
add*	1	1	1	1	**1.000**
sum*	1	1	1	0.5	**0.875**
F					
subtract*	1	1	1	1	**1.000**
difference*	1	0.5	1	0.5	**0.750**
G					
multiply*	1	1	1	1	**1.000**
product*	1	1	1	0.5	**0.875**
H					
divide*	1	1	1	1	**1.000**
division*	1	1	0.5	0.5	**0.750**
quotient*	1	0.5	1	0.5	**0.750**
L					
divides	1	1	0.5	0.5	**0.750**
divisibleBy	1	0.5	1	0.5	**0.750**
isDivisible	1	0	1	1	**0.750**
isDivisibleBy	1	0.5	1	0.5	**0.750**
isDivisor*	0.5	1	0	0.5	**0.500**
modulo	0.5	0	0.5	0.5	**0.375**
M					
gcd*	1	1	1	1	**1.000**
accumulate	0	0	0.5	0.5	**0.250**
modularArithmetic	0	0	0.5	0.5	**0.250**
customOperation	0	0	0	0.5	**0.125**
N					
lcm*	1	1	1	1	**1.000**
complexCalculation	0	0.5	0.5	0.5	**0.375**
complexOperation	0	0.5	0.5	0.5	**0.375**
nestedCalculation	0	0.5	0.5	0.5	**0.375**
accumulateSequence	0	0	0.5	0.5	**0.250**
P					
greaterThan*	1	1	1	1	**1.000**
Q					
greaterThanOrEqual*	1	1	1	1	**1.000**
greaterEqual*	1	1	1	0.5	**0.875**
greaterOrEqual*	1	1	1	0.5	**0.875**
greaterThanOrEqualTo*	1	1	1	0.5	**0.875**
R					
lessThan*	1	1	1	1	**1.000**
S					
lessThanOrEqual*	1	1	1	1	**1.000**
lessEqual*	1	1	1	0.5	**0.875**
lessOrEqual*	1	1	1	0.5	**0.875**
lessThanOrEqualTo*	1	1	1	0.5	**0.875**
T					
equal*	1	1	1	1	**1.000**
equalTo*	1	1	1	0.5	**0.875**
isEqual*	1	1	1	0.5	**0.875**

Table 13.8: LLM judgment results for Case Study 3 (*math*). Judges assign a score of 1 for correct and precise names, 0.5 for imprecise or overly general names that are still correct, and 0 for incorrect names. The final score, shown in the last column, is the average of the individual scores given by the judges. Names that can be considered correct are marked with an asterisk ("*").

Predicate	ChatGPT-5[1]	Gemini	Command R+	**Score**
inv1				
directlyConnected*	1	1	0,5	**0.833**
directConnection*	1	1	0,5	**0.833**
isConnected*	1	0,5	1	**0.833**
can reach	0	0,5	0,5	**0.333**

Table 13.9: LLM judgment results for Case Study 6 (*reachability*). Judges assign a score of 1 for correct and precise names, 0.5 for imprecise or overly general names that are still correct, and 0 for incorrect names. The final score, shown in the last column, is the average of the individual scores given by the judges. Names that can be considered correct are marked with an asterisk ("*").

[1] Since ChatGPT-4o and ChatGPT-o3mini were no longer available at the time of the experiment, ChatGPT-5 was used instead.

Case Study - *Predicate*	**Score**
coauthors - *P*	
coauthorsWithCommonPaper	**0.857**
coauthors	**0.643**
coAuthoredPaper	**0.607**
coauthoredResearchPaper	**0.536**
coAuthorResearchers	**0.464**
authoredTogether	**0.429**
grandparent - *h0*	
parent	**1.000**
renameH0	**0.036**
cousins - *h3*	
siblings	**0.964**
sibling	**0.857**
parent	**0.107**
differentParents	**0.071**
isThirdDegreeRelative	**0.036**
lcm - *G*	
computeProduct	**0.821**
multiply	**0.750**
findLeastCommonMultipleIntermediate	**0.321**
isMultipleOf	**0.214**
lowestCommonMultiple	**0.143**
multiplyAndAdd	**0.143**
greatestCommonDivisor	**0.036**
lcm - *H*	
divide	**0.643**
divideValues	**0.643**
computeLcmFromGcd	**0.500**
lcmCalculation	**0.429**
leastCommonMultiple	**0.357**
divideAndSubtract	**0.143**
highestCommonDivisor	**0.143**

Table 13.10: Human judgment results for Case Studies 1 (*coauthors*) and 4 (*grandparent*, *cousins*, and *lcm*). Judges assign a score of 1 for correct and precise names, 0.5 for imprecise or overly general names that are still correct, and 0 for incorrect names. The final score, shown in the last column, is the average of the individual scores given by the judges. Names that can be considered correct are marked with an asterisk ("*").

Predicate	**Score**
h0	
parent	**1.000**
ancestor	**0.429**
h1	
grandparent	**1.000**
ancestor	**0.393**
h2	
commonAncestor	**0.929**
ancestor	**0.393**
greatAncestor	**0.250**
h3	
sibling	**1.000**
sister	**0.250**
h4	
cousins	**0.893**
cousin	**0.857**
fullSibling	**0.107**
halfSibling	**0.071**
h3	**0.071**

Table 13.11: Human judgment results for Case Study 2 (*family*). Judges assign a score of 1 for correct and precise names, 0.5 for imprecise or overly general names that are still correct, and 0 for incorrect names. The final score, shown in the last column, is the average of the individual scores given by the judges. Names that can be considered correct are marked with an asterisk ("*").

Predicate	**Score**	Predicate	**Score**
A		*L*	
isNumber	**0.929**	isDivisible	**0.464**
numericValue	**0.679**	divides	**0.643**
isInteger	**0.250**	isDivisor	**0.750**
		divisibleBy	**0.393**
B		isDivisibleBy	**0.500**
isEven	**0.929**	modulo	**0.107**
even	**0.750**		
isOdd	**0.071**	*M*	
		gcd	**0.893**
C		modularArithmetic	**0.357**
isOdd	**0.929**	customOperation	**0.286**
odd	**0.750**	accumulate	**0.107**
isEven	**0.071**		
		N	
D		lcm	**0.893**
absoluteValue	**0.964**	nestedCalculation	**0.321**
absValue	**0.964**	complexOperation	**0.286**
negateIfNegative	**0.250**	complexCalculation	**0.286**
isPositiveDifference	**0.000**	accumulateSequence	**0.107**
next	**0.000**		
		P	
E		greaterThan	**1.000**
add	**0.714**		
sum	**0.964**	*Q*	
		greaterThanOrEqual	**0.821**
F		greaterOrEqual	**0.821**
subtract	**0.857**	greaterEqual	**0.500**
difference	**0.893**	greaterThanOrEqualTo	**0.857**
G		*R*	
multiply	**0.786**	lessThan	**0.929**
product	**0.964**		
		S	
H		lessThanOrEqual	**0.821**
divide	**0.750**	lessOrEqual	**0.821**
quotient	**0.679**	lessEqual	**0.500**
division	**0.821**	lessThanOrEqualTo	**0.857**
		T	
		equal	**0.750**
		equalTo	**0.857**
		isEqual	**0.893**

Table 13.12: Human judgment results for Case Study 3 (*math*). Judges assign a score of 1 for correct and precise names, 0.5 for imprecise or overly general names that are still correct, and 0 for incorrect names. The final score, shown in the last column, is the average of the individual scores given by the judges. Names that can be considered correct are marked with an asterisk ("*").

Part V

Conclusions

Chapter 14

Summary and Final Remarks

The work presented in this thesis has explored the development of novel ILP approaches for parameter and structure learning, with the broader goal of integrating logical representation, probabilistic reasoning, and learning mechanisms. After providing the necessary background in Part II, these methods were introduced and discussed in detail in Part III, highlighting how symbolic and probabilistic components can be combined to obtain models that are both expressive and interpretable.

LIFTCOVER+ is an improved version of LIFTCOVER that adds regularization. After testing different configurations of LIFTCOVER+ on several benchmark datasets, our findings confirm that regularization can improve robustness and generalization. These results position LIFTCOVER+ within the broader context of explainable AI, where transparency and structured representations are prioritized over pure scalability.

The remaining approaches address parameter learning in Probabilistic Answer Set Programming using different strategies. Both demonstrated promising performance on the datasets used for evaluation. These results further emphasize the expressive power of PASP as a unifying framework for logic-based probabilistic reasoning, while also highlighting intrinsic scalability challenges related to grounding and answer set enumeration.

Part IV presented a range of applications. LIFTCOVER+ was employed in different domains, yielding mixed results: while Knowledge Graph Completion proved to be a challenging task, its use in an industrial setting was successful, ultimately leading to its integration into the company's internal pipeline. This contrast reflects a recurring theme throughout the thesis: logic-based learning methods are particularly effective

when domain structure and constraints can be exploited, but may face limitations in large-scale, densely connected settings.

We also introduced a Machine Learning pipeline for the discovery of risk factors. Although the overall classification performance was modest, the analysis enabled the identification of potential risk factors for non-adherence to LAI antipsychotics, offering valuable insights to medical specialists. This case study illustrates the practical relevance of interpretable models in sensitive domains such as healthcare, where explainability and interpretability can be as important as predictive accuracy.

Furthermore, we contributed two datasets to the Neuro-Symbolic AI community, expanding the availability of benchmarks for evaluating learning systems that combine logic and probability. Such resources are intended to support future research on NeSy approaches.

Finally, we investigated the integration of LLMs and logic to improve the readability and interpretability of logic theories containing unnamed predicates. The experimental results indicate that LLMs offer a promising direction for tackling this challenge, especially considering their continuously improving capabilities. Given their ability to interpret code and produce natural language explanations, future work could explore their use in enhancing the explainability of machine learning models by generating human-readable explanations for model outputs.

Overall, this thesis highlights both the potential and the limitations of logic-based probabilistic learning. While scalability remains a fundamental challenge due to factors such as grounding overhead and the cost of symbolic inference, these limitations represent the price paid for high interpretability, rich relational structure, and explicit reasoning. Future research directions include improving scalability through more efficient representations, exploring distributed implementations of learning algorithms, and further investigating the integration of symbolic reasoning with data-driven models such as LLMs.

Author's Publication List

1. E. Gentili, A. Bizzarri, D. Azzolini, and F. Riguzzi, "The gradient semiring for probabilistic answer set programming and its application to parameter learning," in *Proceedings of the 5th International Joint Conference on Learning & Reasoning*, 2025. To appear.

2. E. Gentili, T. Ribeiro, F. Riguzzi, and K. Inoue, "Predicate renaming via large language models," *arXiv preprint arXiv:2510.25517*, 2025.

3. E. Gentili, G. Franchini, R. Zese, M. Alberti, M. Ferrara, I. Domenicano, and L. Grassi, "Machine learning from real data: A mental health registry case study," *Computer Methods and Programs in Biomedicine Update*, vol. 5, p. 100132, 2024. https://doi.org/10.1016/j.cmpbup.2023.100132

4. E. Gentili, A. Bizzarri, D. Azzolini, R. Zese, and F. Riguzzi, "Regularization in probabilistic inductive logic programming," in *International Conference on Inductive Logic Programming*, pp. 16–29, Springer, 2023. doi:10.1007/978-3-031-49299-0_2
   Best Student Paper Award at the 32nd International Conference on Inductive Logic Programming ILP2023 @ IJCLR.

5. E. Gentili, "Knowledge graph completion with probabilistic logic programming," *Proceedings of the AIxIA Doctoral Consortium*, 2023. https://ceur-ws.org/Vol-3670/paper96.pdf

6. D. Azzolini, E. Gentili, and F. Riguzzi, "Symbolic parameter learning in probabilistic answer set programming," *Theory and Practice of Logic Programming*, vol. 24, no. 4, pp. 698–715, 2024. doi:10.1017/S1471068424000334

7. D. Azzolini, M. Bonato, E. Gentili, F. Riguzzi, "Logic programming for knowledge graph completion," *39th Italian Conference on Computational Logic (CILC 2024)*, in *CEUR Workshop Proceedings*, vol. 3733, pp. 1–14, CEUR-WS, 2024. https://ceur-ws.org/Vol-3733/paper4.pdf

8. R. Manhaeve, F. Giannini, M. Ali, D. Azzolini, A. Bizzarri, A. Borghesi, S. Bortolotti, L. De Raedt, D. Dhami, M. Diligenti, S. Dumančić, B. Faltings, E. Gentili, A. Gerevini, M. Gori, T. Guns, M. Homola, K. Kersting, J. Lehmann, M. Lombardi, L. Lorello, E. Marconato, S. Melacci, A. Passerini, D. Paul, F. Riguzzi, S. Teso, N. Yorke-Smith, and M. Lippi, "Benchmarking in neuro-symbolic ai," in *Learning and Reasoning: 4th International Joint Conference on Learning and Reasoning, IJCLR 2024, and 33rd International Conference on Inductive Logic Programming, ILP 2024, Nanjing, China, September 20–22, 2024, Proceedings*, W.-Z. Dai, Ed. Cham: Springer Nature Switzerland, 2026, pp. 238–249.

9. D. Azzolini, E. Gentili, F. Riguzzi, "Link prediction in knowledge graphs with probabilistic logic programming: Work in progress," *Workshop on Probabilistic Logic Programming (PLP 2023)*, in *CEUR Workshop Proceedings*, vol. 3437, pp. 1–4, CEUR-WS, 2023. https://ceur-ws.org/Vol-3437/short5PLP.pdf

10. M. Ferrara, E. Gentili, M. B. Murri, R. Zese, M. Alberti, G. Franchini, I. Domenicano, F. Folesani, C. Sorio, L. Benini, P. Carozza, J. Little, L. Grassi, "Establishment of a public mental health database for research purposes in the Ferrara province: Development and preliminary evaluation study," *JMIR Medical Informatics*, vol. 11, no. 1, p. e45523, 2023. doi:10.2196/45523

11. M. Ferrara, E. M. A. Curtarello, E. Gentili, I. Domenicano, L. Vecchioni, R. Zese, M. Alberti, G. Franchini, C. Sorio, L. Benini, J. Little, P. Carozza, P. Dazzan, and L. Grassi , "Sex differences in schizophrenia spectrum diagnoses: results from a 30-year health record registry," *Archives of women's mental health*, vol. 27, no. 1, pp. 11–20, 2024. https://doi.org/10.1007/s00737-023-01371-8

12. M. Ferrara, I. Domenicano, A. Bellagamba, G. Zaffarami, L. Benini, C. Sorio, E. Gentili, V. H. Srihari, and L. Grassi, "Sex differences in clozapine prescription: Results from an Italian 30-year health records registry," *Journal of Psychiatric Research*, vol. 185, pp. 215–223, 2025. https://doi.org/10.1016/j.jpsychires.2025.02.019

13. M. Ferrara, I. Domenicano, A. Bellagamba, G. Zaffarami, A. Onofrio, E. Gentili, L. Benini, C. Sorio, F. Emanuelli, "Sex differences in clozapine prescription: results from a 30-year retrospective study in an Italian province.," *Neuroscience Applied*, vol. 3, p. 105055, 2024.
https://doi.org/10.1016/j.nsa.2024.105055

14. M. Ferrara, I. Domenicano, A. Marchi, G. Zaffarami, A. Onofrio, L. Benini, C. Sorio, E. Gentili, M. B. Murri, T. Toffanin, J. Little, and L. Grassi, "First episode psychoses in people over-35 years old: uncovering potential actionable targets for early intervention services," *Psychiatry Research*, vol. 339, p. 116034, 2024. https://doi.org/10.1016/j.psychres.2024.116034

Bibliography

[1] T. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997.

[2] T. Hastie, R. Tibshirani, J. Friedman *et al.*, *The elements of statistical learning*. Springer series in statistics New-York, 2009.

[3] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[4] R. S. Sutton, A. G. Barto *et al.*, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[5] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[6] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[7] ——, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[8] ——, "Improved use of continuous attributes in c4. 5," *Journal of artificial intelligence research*, vol. 4, pp. 77–90, 1996.

[9] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. Chapman and Hall/CRC, 2017.

[10] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011.

[11] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[12] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[13] J. Zhu, H. Zou, S. Rosset, T. Hastie *et al.*, "Multi-class adaboost," *Statistics and its Interface*, vol. 2, no. 3, pp. 349–360, 2009.

[14] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[15] C. M. Bishop and H. Bishop, *Deep learning: Foundations and concepts*. Springer Nature, 2023.

[16] Y. Bengio, I. Goodfellow, A. Courville *et al.*, *Deep learning*. MIT press Cambridge, MA, USA, 2017, vol. 1.

[17] S. Feuerriegel, J. Hartmann, C. Janiesch, and P. Zschech, "Generative ai," *Business & Information Systems Engineering*, vol. 66, no. 1, pp. 111–126, 2024.

[18] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[19] Y. Liu, H. He, T. Han, X. Zhang, M. Liu, J. Tian, Y. Zhang, J. Wang, X. Gao, T. Zhong *et al.*, "Understanding llms: A comprehensive overview from training to inference," *arXiv preprint arXiv:2401.02038*, 2024.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, Inc., 2017, vol. 30. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[21] J. Alammar and M. Grootendorst, *Hands-On Large Language Models*. Sebastopol, CA, USA: O'Reilly, 2024. [Online]. Available: https://www.oreilly.com/library/view/hands-on-large-language/9781098150952/

[22] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[23] Y. Yang, L. Wang, S. Shi, P. Tadepalli, S. Lee, and Z. Tu, "On the sub-layer functionalities of transformer decoder," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 4799–4811. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.432/

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds. Red Hook, NY, USA: Curran Associates, Inc., 2020, vol. 33, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[25] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," *arXiv preprint arXiv:2307.10169*, 2023.

[26] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[27] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2339–2356.

[28] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?"

Journal of Systems and Software, vol. 203, p. 111734, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001292

[29] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[30] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, " Rethinking the Role of Demonstrations: What makes In-context Learning Work? ," in *EMNLP*, 2022.

[31] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[32] T. Wu, M. Terry, and C. J. Cai, "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts," in *Proceedings of the 2022 CHI conference on human factors in computing systems*, 2022, pp. 1–22.

[33] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[34] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Red Hook, NY, USA: Curran Associates, Inc., 2022, pp. 22 199–22 213. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf

[35] OpenAI, "OpenAI o3-mini System Card," 2025. [Online]. Available: https://openai.com/index/o3-mini-system-card

[36] J. W. Lloyd, *Foundations of logic programming*. Springer Science & Business Media, 2012.

[37] F. Riguzzi, *Foundations of Probabilistic Logic Programming Languages, Semantics, Inference and Learning, Second Edition*. Gistrup, Denmark: River Publishers, 2023.

[38] R. A. Kowalski, "The early years of logic programming," *Commun. ACM*, vol. 31, pp. 38–43, 1988. [Online]. Available: https://api.semanticscholar.org/CorpusID:12259230

[39] A. Colmerauer and P. Roussel, *The birth of Prolog*. New York, NY, USA: Association for Computing Machinery, 1996, p. 331–367. [Online]. Available: https://doi.org/10.1145/234286.1057820

[40] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Reading, MA, USA: Addison-Wesley, 1995.

[41] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *ICLP/SLP*, 1988. [Online]. Available: https://api.semanticscholar.org/CorpusID:261517573

[42] R. Kowalski, "Predicate logic as programming language," in *IFIP congress*, vol. 74, 1974, pp. 569–544.

[43] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965.

[44] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming." in *5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, vol. 88. USA: MIT Press, 1988, pp. 1070–1080.

[45] A. Van Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs," *Journal of the ACM*, vol. 38, no. 3, pp. 620–650, 1991.

[46] T. Sato, "A statistical learning method for logic programs with distribution semantics," in *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, L. Sterling, Ed. MIT Press, 1995, pp. 715–729.

[47] L. De Raedt and A. Kimmig, "Probabilistic (logic) programming concepts," *Machine Learning*, vol. 100, no. 1, pp. 5–47, 2015.

[48] S. Mørk and I. Holmes, "Evaluating bacterial gene-finding hmm structures as probabilistic logic programs," *Bioinformatics*, vol. 28, no. 5, pp. 636–642, 2012.

[49] A. Nguembang Fadja and F. Riguzzi, "Probabilistic logic programming in action," in *Towards Integrative Machine Learning and Knowledge Extraction*, ser. Lecture Notes in Computer Science, A. Holzinger, R. Goebel, M. Ferri, and V. Palade, Eds. Springer, 2017, vol. 10344, pp. 89–116.

[50] F. Riguzzi, E. Lamma, M. Alberti, E. Bellodi, R. Zese, and G. Cota, "Probabilistic logic programming for natural language processing," in *Workshop on Deep Understanding and Reasoning, URANIA 2016*, ser. CEUR Workshop Proceedings, F. Chesani, P. Mello, and M. Milano, Eds., vol. 1802. Sun SITE Central Europe, 2017, pp. 30–37.

[51] F. Riguzzi and T. Swift, "Probabilistic logic programming under the distribution semantics," in *Declarative Logic Programming: Theory, Systems, and Applications*, M. Kifer and Y. A. Liu, Eds. Association for Computing Machinery and Morgan & Claypool, 2018.

[52] E. Dantsin, "Probabilistic logic programs and their semantics," in *Logic Programming: First Russian Conference on Logic Programming Irkutsk, Russia, September 14–18, 1990 Second Russian Conference on Logic Programming St. Petersburg, Russia, September 11–16, 1991 Proceedings*. Springer, 2005, pp. 152–164.

[53] J. Vennekens, S. Verbaeten, and M. Bruynooghe, "Logic programs with annotated disjunctions," in *Logic Programming: 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004. Proceedings 20*. Springer, 2004, pp. 431–445.

[54] S. Taisuke, "A statistical learning method for logic programs with distribution semantics," in *Proceedings of ICLP*, 1995, pp. 715–729.

[55] L. De Raedt, A. Kimmig, and H. Toivonen, "Problog: A probabilistic prolog and its application in link discovery," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, ser. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, p. 2468–2473.

[56] J. Vennekens, S. Verbaeten, and M. Bruynooghe, "Logic Programs With Annotated Disjunctions," in *20th International Conference on Logic Programming (ICLP 2004)*, ser. Lecture Notes in Computer Science, vol. 3132. Springer, 2004, pp. 431–445.

[57] A. S. d. Garcez, K. B. Broda, and D. M. Gabbay, *Neural-symbolic learning systems*. Springer, 2002.

[58] T. R. Besold, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K.-U. Kühnberger, L. C. Lamb, P. M. V. Lima, L. de Penning, G. Pinkas *et al.*, "Neural-symbolic learning and reasoning: A survey and interpretation 1," in *Neuro-symbolic artificial intelligence: The state of the art*. IOS press, 2021, pp. 1–51.

[59] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt, "Deepproblog: Neural probabilistic logic programming," *Advances in neural information processing systems*, vol. 31, 2018.

[60] L. De Raedt, R. Manhaeve, S. Dumancic, T. Demeester, and A. Kimmig, "Neuro-symbolic= neural+ logical+ probabilistic." in *NeSy@ IJCAI*, 2019.

[61] R. Reiter, "On closed world data bases," in *Readings in artificial intelligence*. Elsevier, 1981, pp. 119–140.

[62] J. R. Quinlan, "Learning logical definitions from relations," *Machine learning*, vol. 5, no. 3, pp. 239–266, 1990.

[63] S. Muggleton, "Inverse entailment and progol," *New generation computing*, vol. 13, no. 3, pp. 245–286, 1995.

[64] L. De Raedt and W. Van Laer, "Inductive constraint logic," in *International Workshop on Algorithmic Learning Theory*. Springer, 1995, pp. 80–94.

[65] A. Cropper and S. Dumančić, "Inductive logic programming at 30: a new introduction," *Journal of Artificial Intelligence Research*, vol. 74, pp. 765–850, 2022.

[66] G. D. Plotkin, "A note on inductive generalization," *Machine intelligence*, vol. 5, no. 1, pp. 153–163, 1970.

[67] L. De Raedt and K. Kersting, "Probabilistic inductive logic programming," in *International conference on algorithmic learning theory*. Springer, 2004, pp. 19–36.

[68] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt, "Inference and learning in probabilistic logic programs using weighted boolean formulas," *Theory and Practice of Logic Programming*, vol. 15, no. 3, pp. 358–401, 2015.

[69] E. Bellodi and F. Riguzzi, "Expectation maximization over binary decision diagrams for probabilistic logic programs," *Intelligent Data Analysis*, vol. 17, no. 2, pp. 343–363, 2013.

[70] L. De Raedt and I. Thon, "Probabilistic rule learning," in *International conference on inductive logic programming*. Springer, 2010, pp. 47–58.

[71] L. De Raedt, A. Dries, I. Thon, G. Van den Broeck, M. Verbeke, Q. Yang, and M. Wooldridge, "Inducing probabilistic relational rules from probabilistic examples," in *Proceedings of 24th international joint conference on artificial intelligence (IJCAI)*, vol. 2015. IJCAI-INT JOINT CONF ARTIF INTELL, 2015, pp. 1835–1842.

[72] E. Bellodi and F. Riguzzi, "Structure learning of probabilistic logic programs by searching the clause space," *Theory and Practice of Logic Programming*, vol. 15, no. 2, pp. 169–212, 2015.

[73] D. Poole, "First-order probabilistic inference," in *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann Publishers, 2003, pp. 985–991.

[74] A. Nguembang Fadja and F. Riguzzi, "Lifted discriminative learning of probabilistic logic programs," *Machine Learning*, vol. 108, no. 7, pp. 1111–1135, 2019.

[75] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society: Series B*, vol. 39, pp. 1–38, 1977.

[76] A. Nguembang Fadja, F. Riguzzi, and E. Lamma, "Learning hierarchical probabilistic logic programs," *Machine Learning*, vol. 110, no. 7, pp. 1637–1693, 2021.

[77] S. Kok and P. Domingos, "Learning the structure of Markov Logic Networks," in *22nd International Conference on Machine learning*. ACM, 2005, pp. 441–448.

[78] O. Schulte and H. Khosravi, "Learning graphical models for relational data via lattice search," *Machine Learning*, vol. 88, no. 3, pp. 331–368, 2012.

[79] A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg, "Carcinogenesis predictions using ILP," in *7th International Workshop on Inductive Logic Programming*, ser. Lecture Notes in Computer Science, N. Lavrac and S. Džeroski, Eds., vol. 1297. Springer Berlin Heidelberg, 1997, pp. 273–287.

[80] A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King, "Theories for mutagenicity: A study in first-order and feature-based induction," *Artificial Intelligence*, vol. 85, no. 1-2, pp. 277–299, 1996.

[81] J. Struyf, J. Davis, and D. Page, "An efficient approximation to lookahead in relational learners," in *European Conference on Machine Learning (ECML 2006)*, ser. Lecture Notes in Computer Science. Springer, 2006, pp. 775–782.

[82] J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in *European Conference on Machine Learning (ECML 2006)*. ACM, 2006, pp. 233–240.

[83] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, pp. 861–874, 2006.

[84] F. Riguzzi, E. Bellodi, R. Zese, G. Cota, and E. Lamma, "A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics," *International Journal of Approximate Reasoning*, vol. 80, pp. 313–333, 1 2017.

[85] A. Kimmig, L. Mihalkova, and L. Getoor, "Lifted graphical models: a survey," *Machine Learning*, vol. 99, pp. 1–45, 2015.

[86] E. Bellodi and F. Riguzzi, "Structure learning of probabilistic logic programs by searching the clause space," *Theory and Practice of Logic Programming*, vol. 15, no. 2, pp. 169–212, 2015.

[87] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 509–516, 1978.

[88] L. D. Raedt, A. Dries, I. Thon, G. V. den Broeck, and M. Verbeke, "Inducing probabilistic relational rules from probabilistic examples," in *24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, Q. Yang and M. Wooldridge, Eds. AAAI Press, 2015, pp. 1835–1843.

[89] F. G. Cozman and D. D. Mauá, "The structure and complexity of credal semantics," in *3rd International Workshop on Probabilistic Logic Programming (PLP 2016)*, ser. CEUR Workshop Proceedings, A. Hommersom and S. A. Abdallah, Eds., vol. 1661. CEUR-WS.org, 2016, pp. 3–14.

[90] D. D. Mauá and F. G. Cozman, "Complexity results for probabilistic answer set programming," *International Journal of Approximate Reasoning*, vol. 118, pp. 133–154, 2020.

[91] T. Lukasiewicz, "Probabilistic description logic programs," in *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 8th European Conference, ECSQARU 2005, Barcelona, Spain, July 6-8, 2005, Proceedings*, ser. Lecture Notes in Computer Science, L. Godo, Ed., vol. 3571. Springer, 2005, pp. 737–749.

[92] L. De Raedt, A. Kimmig, and H. Toivonen, "ProbLog: A probabilistic Prolog and its application in link discovery," in *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, M. M. Veloso, Ed., vol. 7. AAAI Press, 2007, pp. 2462–2467.

[93] F. G. Cozman and D. D. Mauá, "The joy of probabilistic answer set programming: Semantics, complexity, expressivity, inference," *International Journal of Approximate Reasoning*, vol. 125, pp. 218–239, 2020.

[94] R. Kiesel, P. Totis, and A. Kimmig, "Efficient knowledge compilation beyond weighted model counting," *Theory and Practice of Logic Programming*, vol. 22, no. 4, pp. 505–522, 2022.

[95] A. Kimmig, G. Van den Broeck, and L. De Raedt, "Algebraic model counting," *J. of Applied Logic*, vol. 22, no. C, pp. 46–62, Jul. 2017.

[96] D. Azzolini and F. Riguzzi, "Inference in probabilistic answer set programming under the credal semantics," in *AIxIA 2023 - Advances in Artificial Intelligence*, ser. Lecture Notes in Artificial Intelligence, R. Basili, D. Lembo, C. Limongelli, and A. Orlandini, Eds., vol. 14318. Heidelberg, Germany: Springer, 2023, pp. 367–380.

[97] A. Darwiche and P. Marquis, "A knowledge compilation map," *Journal of Artificial Intelligence Research*, vol. 17, pp. 229–264, 2002.

[98] T. Eiter, M. Hecher, and R. Kiesel, "Treewidth-aware cycle breaking for algebraic answer set counting," in *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021*, M. Bienvenu, G. Lakemeyer, and E. Erdem, Eds., 2021, pp. 269–279.

[99] Eiter, Thomas and Hecher, Markus and Kiesel, Rafael, "aspmc: New frontiers of algebraic answer set counting," *Artificial Intelligence*, vol. 330, p. 104109, 2024.

[100] D. Azzolini, E. Bellodi, and F. Riguzzi, "Statistical statements in probabilistic logic programming," in *Logic Programming and Nonmonotonic Reasoning*, G. Gottlob, D. Inclezan, and M. Maratea, Eds. Cham: Springer International Publishing, 2022, pp. 43–55.

[101] Azzolini, Damiano and Bellodi, Elena and Riguzzi, Fabrizio, "Learning the parameters of probabilistic answer set programs," in *Inductive Logic Programming*, S. H. Muggleton and A. Tamaddoni-Nezhad, Eds. Cham: Springer Nature Switzerland, 2024, pp. 1–14.

[102] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas,

D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[103] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "SymPy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017.

[104] M. J. D. Powell, "A direct search optimization method that models the objective and constraint functions by linear interpolation," in *Advances in Optimization and Numerical Analysis*, S. Gomez and J.-P. Hennart, Eds. Dordrecht: Springer Netherlands, 1994, pp. 51–67.

[105] D. Kraft, "Algorithm 733: TOMP-fortran modules for optimal control calculations," *ACM Transactions on Mathematical Software*, vol. 20, no. 3, pp. 262–281, Sep. 1994.

[106] P. Totis, L. De Raedt, and A. Kimmig, "smProbLog: Stable model semantics in ProbLog for probabilistic argumentation," *Theory and Practice of Logic Programming*, pp. 1–50, 2023.

[107] E. Bellodi and F. Riguzzi, "Expectation maximization over binary decision diagrams for probabilistic logic programs," *Intelligent Data Analysis*, vol. 17, no. 2, pp. 343–363, 2013.

[108] D. Azzolini, "A constrained optimization approach to set the parameters of probabilistic answer set programs," in *Inductive Logic Programming*, E. Bellodi, F. A. Lisi, and R. Zese, Eds. Cham: Springer Nature Switzerland, 2023, pp. 1–15.

[109] D. Azzolini, E. Bellodi, and F. Riguzzi, "Learning the parameters of probabilistic answer set programs," in *Inductive Logic Programming*, S. H. Muggleton and A. Tamaddoni-Nezhad, Eds. Cham: Springer Nature Switzerland, 2024, pp. 1–14.

[110] D. Azzolini, E. Gentili, and F. Riguzzi, "Symbolic parameter learning in probabilistic answer set programming," *Theory and Practice of Logic Programming*, vol. 24, no. 4, pp. 698–715, 2024.

[111] B. Gutmann, A. Kimmig, K. Kersting, and L. D. Raedt, "Parameter learning in probabilistic databases: A least squares approach," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2008)*, ser. Lecture Notes in Computer Science, vol. 5211. Springer, 2008, pp. 473–488.

[112] T. Eiter, M. Hecher, and R. Kiesel, "aspmc: New frontiers of algebraic answer set counting," *Artificial Intelligence*, vol. 330, p. 104109, 2024.

[113] A. Darwiche, "Decomposable negation normal form," *Journal of the ACM*, vol. 48, no. 4, pp. 608–647, 2001.

[114] K. R. Apt and R. N. Bol, "Logic programming and negation: A survey," *The Journal of Logic Programming*, vol. 19, pp. 9–71, 1994.

[115] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo," *Theory and Practice of Logic Programming*, vol. 19, no. 1, pp. 27–82, 2019.

[116] E. Gentili, A. Bizzarri, D. Azzolini, R. Zese, and F. Riguzzi, "Regularization in probabilistic inductive logic programming," in *International Conference on Inductive Logic Programming*. Springer, 2023, pp. 16–29.

[117] D. Fierens, G. Van den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt, "Inference and learning in probabilistic logic programs using weighted Boolean formulas," *Theory and Practice of Logic Programming*, vol. 15, no. 3, pp. 358–401, 2015.

[118] B. Gutmann, I. Thon, and L. De Raedt, "Learning the parameters of probabilistic logic programs from interpretations," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2011)*, ser. Lecture Notes in Computer Science, D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds., vol. 6911. Springer, 2011, pp. 581–596.

[119] R. L. Geh, J. Goncalves, I. C. Silveira, D. D. Maua, and F. G. Cozman, "dPASP: A comprehensive differentiable probabilistic answer set programming environment for neurosymbolic learning and reasoning," *arXiv*, 2023.

[120] C. Baral, M. Gelfond, and N. Rushton, "Probabilistic reasoning with answer sets," *Theory and Practice of Logic Programming*, vol. 9, no. 1, pp. 57–144, 2009.

[121] J. Lee and Y. Wang, "Weighted rules under the stable model semantics," in *Proceedings of the Fifteenth International Conference on Principles of Knowledge Representation and Reasoning*, C. Baral, J. P. Delgrande, and F. Wolter, Eds. AAAI Press, 2016, pp. 145–154.

[122] J. Lee and Z. Yang, "LPMLN, weak constraints, and P-log," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 1170–1177.

[123] Z. Chen, Y. Wang, B. Zhao, J. Cheng, X. Zhao, and Z. Duan, "Knowledge graph completion: A review," *IEEE Access*, vol. 8, pp. 192 435–192 456, 2020.

[124] T. Shen, F. Zhang, and J. Cheng, "A comprehensive overview of knowledge graph completion," *Knowledge-Based Systems*, vol. 255, p. 109597, 2022.

[125] C. Meilicke, M. W. Chekol, D. Ruffinelli, and H. Stuckenschmidt, "Anytime bottom-up rule learning for knowledge graph completion." in *IJCAI*, 2019, pp. 3137–3143.

[126] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," *Advances in neural information processing systems*, vol. 26, 2013.

[127] S. Ott, C. Meilicke, and M. Samwald, "Safran: An interpretable, rule-based link prediction method outperforming embedding models," *arXiv preprint arXiv:2109.08002*, 2021.

[128] R. Zhang, Y. Mao, and W. Zhao, "Knowledge graphs completion via probabilistic reasoning," *Information Sciences*, vol. 521, pp. 144–159, 2020.

[129] L. A. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek, "AMIE: association rule mining under incomplete evidence in ontological knowledge bases," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 413–422.

[130] Z. Wang, J. Zhang, J. Feng, and Z. Chen, "Knowledge graph embedding by translating on hyperplanes," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 28, 2014.

[131] D. Azzolini and F. Riguzzi, "Optimizing probabilities in probabilistic logic programs," *Theory and Practice of Logic Programming*, vol. 21, no. 5, pp. 543–556, 2021.

[132] C. Meilicke, M. W. Chekol, P. Betz, M. Fink, and H. Stuckeschmidt, "Anytime bottom-up rule learning for large-scale knowledge graph completion," *The VLDB Journal*, vol. 33, no. 1, pp. 131–161, 2024.

[133] C. Meilicke, "Personal communication," 2024.

[134] P. Betz, C. Meilicke, S. Ott, and L. Galárraga, "PyClause," 2024. [Online]. Available: https://github.com/symbolic-kg/PyClause/

[135] K. Toutanova and D. Chen, "Observed versus latent features for knowledge base and text inference," in *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*, A. Allauzen, E. Grefenstette, K. M. Hermann, H. Larochelle, and S. W.-t. Yih, Eds. Beijing, China: Association for Computational Linguistics, Jul. 2015, pp. 57–66.

[136] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013.

[137] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka, and T. Mitchell, "Toward an architecture for never-ending language learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 24, 2010, pp. 1306–1313.

[138] R. J. Rummel, "Dimensionality of Nations Project: Attributes of Nations and Behavior of Nation Dyads, 1950-1965," 1992.

[139] Istituto Nazionale di Statistica, "Population census and dynamics - year 2020," 2021. [Online]. Available: https://www.istat.it/it/archivio/264511

[140] M. Ferrara, E. Gentili, M. B. Murri, R. Zese, M. Alberti, G. Franchini, I. Domenicano, F. Folesani, C. Sorio, L. Benini, P. Carozza, J. Little, and L. Grassi, "Establishment of a public mental health database for research purposes in the ferrara province: Development and preliminary evaluation study," *JMIR Medical Informatics*, vol. 11, no. 1, p. e45523, 2023.

[141] E. Gentili, G. Franchini, R. Zese, M. Alberti, M. Ferrara, I. Domenicano, and L. Grassi, "Machine learning from real data: A mental health registry case study," *Computer Methods and Programs in Biomedicine Update*, vol. 5, p. 100132, 2024.

[142] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[143] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[144] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *the Journal of machine Learning research*, vol. 9, pp. 1871–1874, 2008.

[145] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[146] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[147] S. Seabold and J. Perktold, "statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010.

[148] J. Chung and J. Teo, "Mental health prediction using machine learning: taxonomy, applications, and challenges," *Applied Computational Intelligence and Soft Computing*, vol. 2022, no. 1, p. 9970363, 2022.

[149] A. Bohlmann, J. Mostafa, and M. Kumar, "Machine learning and medication adherence: scoping review," *JMIRx med*, vol. 2, no. 4, p. e26993, 2021.

[150] Z. Zhu, D. Roy, S. Feng, and B. Vogler, "Ai-based medication adherence prediction in patients with schizophrenia and attenuated psychotic disorders," *Schizophrenia Research*, vol. 275, pp. 42–51, 2025.

[151] Y. Gu, A. Zalkikar, M. Liu, L. Kelly, A. Hall, K. Daly, and T. Ward, "Predicting medication adherence using ensemble learning and deep learning models with large scale healthcare data," *Scientific Reports*, vol. 11, no. 1, p. 18961, 2021.

[152] C. Molnar, *Interpretable machine learning*. Lulu. com, 2020.

[153] M. Mozaffari-Kermani, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, "Systematic poisoning attacks on and defenses for machine learning in healthcare," *IEEE journal of biomedical and health informatics*, vol. 19, no. 6, pp. 1893–1905, 2014.

[154] M. A. Ahmad, C. Eckert, and A. Teredesai, "Interpretable machine learning in healthcare," in *Proceedings of the 2018 ACM international conference on bioinformatics, computational biology, and health informatics*, 2018, pp. 559–560.

[155] W. Stammer, P. Schramowski, and K. Kersting, "Right for the right concept: Revising neuro-symbolic concepts by interacting with their explanations," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 3619–3629.

[156] H. Müller and A. Holzinger, "Kandinsky patterns," *Artificial intelligence*, vol. 300, p. 103546, 2021.

[157] E. Marconato, G. Bontempo, E. Ficarra, S. Calderara, A. Passerini, and S. Teso, "Neuro-symbolic continual learning: Knowledge, reasoning shortcuts and concept rehearsal," *arXiv preprint arXiv:2302.01242*, 2023.

[158] A. Srinivasan, S. H. Muggleton, M. J. Sternberg, and R. D. King, "Theories for mutagenicity: A study in first-order and feature-based induction," *Artificial Intelligence*, vol. 85, no. 1-2, pp. 277–299, 1996.

[159] A. Srinivasan, R. D. King, S. H. Muggleton, and M. J. Sternberg, "Carcinogenesis predictions using ilp," in *International Conference on Inductive Logic Programming*. Springer, 1997, pp. 273–287.

[160] K. Inoue, A. Doncescu, and H. Nabeshima, "Completing causal networks by meta-level abduction," *Machine learning*, vol. 91, no. 2, pp. 239–277, 2013.

[161] R. J. Mooney, "Inductive logic programming for natural language processing," in *International conference on inductive logic programming*. Springer, 1996, pp. 1–22.

[162] M. Law, A. Russo, and K. Broda, "Inductive learning of answer set programs," in *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings 14*. Springer, 2014, pp. 311–325.

[163] K. Basu, K. Murugesan, M. Atzeni, P. Kapanipathi, K. Talamadupula, T. Klinger, M. Campbell, M. Sachan, and G. Gupta, "A hybrid neuro-symbolic approach for text-based games using inductive logic programming," in *Combining learning and reasoning: programming languages, formalisms, and representations*, 2021.

[164] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[165] Z. Zhang, L. Yilmaz, and B. Liu, "A critical review of inductive logic programming techniques for explainable ai," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 8, pp. 10 220–10 236, 2024.

[166] K. Inoue, T. Ribeiro, and C. Sakama, "Learning from interpretation transition," *Machine Learning*, vol. 94, pp. 51–79, 2014.

[167] A. Cropper and R. Morel, "Predicate invention by learning from failures," *arXiv preprint arXiv:2104.14426*, 2021.

[168] A. Nguembang Fadja, F. Riguzzi, and E. Lamma, "Learning hierarchical probabilistic logic programs," *Machine Learning*, vol. 110, no. 7, pp. 1637–1693, 2021.

[169] J. Sha, H. Shindo, K. Kersting, and D. S. Dhami, "Neuro-symbolic predicate invention: Learning relational concepts from visual scenes," *Neurosymbolic Artificial Intelligence*, no. Preprint, pp. 1–26, 2024.

[170] S. Muggleton, "Inductive logic programming," *New generation computing*, vol. 8, pp. 295–318, 1991.

[171] S. Muggleton and W. Buntine, "Machine invention of first-order predicates by inverting resolution," in *Machine Learning Proceedings 1988*. San Francisco (CA): Elsevier, 1988, pp. 339–352.

[172] I. Stahl, "Predicate invention in ilp—an overview," in *European conference on machine learning*. Springer, 1993, pp. 311–322.

[173] ——, "The appropriateness of predicate invention as bias shift operation in ilp," *Machine Learning*, vol. 20, pp. 95–117, 1995.

[174] S. Muggleton, L. De Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan, "Ilp turns 20: biography and future challenges," *Machine learning*, vol. 86, pp. 3–23, 2012.

[175] S. Kramer, "Predicate invention: A comprehensive view," *Rapport technique OFAI-TR-95-32, Austrian Research Institute for Artificial Intelligence, Vienna*, 1995.

[176] S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad, "Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited," *Machine Learning*, vol. 100, no. 1, pp. 49–73, 2015.

[177] A. Cropper and R. Morel, "Learning programs by learning from failures," *Machine Learning*, vol. 110, no. 4, pp. 801–856, 2021.

[178] F. Riguzzi, *Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning (2nd ed.)*. Gistrup, Denmark: River Publishers, 2023.

[179] A. N. Fadja, E. Lamma, and F. Riguzzi, "Deep probabilistic logic programming." in *Plp@ Ilp*, 2017, pp. 3–14.

[180] S. Kok and P. Domingos, "Learning the structure of markov logic networks," in *Proceedings of the 22nd international conference on Machine learning*, 2005, pp. 441–448.

[181] A. C. Kakas, R. A. Kowalski, and F. Toni, "Abductive logic programming," *Journal of logic and computation*, vol. 2, no. 6, pp. 719–770, 1992.

[182] T. Sato, K. Inoue, and C. Sakama, "Abducing relations in continuous spaces." in *IJCAI*, 2018, pp. 1956–1962.

[183] OpenAI, "GPT-4o System Card," 2024. [Online]. Available: https://openai.com/index/gpt-4o-system-card

[184] ——, "Hello GPT-4o," 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o

[185] Meta, "Llama 3.2: Revolutionizing edge AI and vision with open, customizable models," 2024. [Online]. Available: https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md

[186] ——, "Llama 3.2 Model Card," 2024. [Online]. Available: https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md

[187] G. DeepMind, "Gemini 1.5 Flash," 2024. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini?hl=it#gemini-1.5-flash

[188] J. Zuo, M. Velikanov, D. E. Rhaiem, I. Chahed, Y. Belkada, G. Kunsch, and H. Hacid, "Falcon mamba: The first competitive attention-free 7b language model," 2024. [Online]. Available: https://arxiv.org/abs/2410.05355

[189] Falcon-LLM Team, "The Falcon 3 Family of Open Models," December 2024. [Online]. Available: https://huggingface.co/blog/falcon3

[190] Cohere For AI, "c4ai-command-r-plus-08-2024," 2024. [Online]. Available: https://huggingface.co/CohereForAI/c4ai-command-r-plus-08-2024

[191] A. Koubaa, W. Boulila, L. Ghouti, A. Alzahem, and S. Latif, "Exploring chatgpt capabilities and limitations: a survey," *IEEE Access*, 2023.

[192] K. Jeblick, B. Schachtner, J. Dexl, A. Mittermeier, A. T. Stüber, J. Topalis, T. Weber, P. Wesp, B. O. Sabel, J. Ricke *et al.*, "Chatgpt makes medicine easy to swallow: an exploratory case study on simplified radiology reports," *European radiology*, vol. 34, no. 5, pp. 2817–2825, 2024.

[193] A. Rao, M. Pang, J. Kim, M. Kamineni, W. Lie, A. K. Prasad, A. Landman, K. Dreyer, and M. D. Succi, "Assessing the utility of chatgpt throughout the entire clinical workflow: development and usability study," *Journal of Medical Internet Research*, vol. 25, p. e48659, 2023.

[194] A. Dash, B. Mathur, and P. Patil, "Enhancing carbon emission tracking with gemini: An ai solution for sustainable development," 2024.

[195] G. Mondillo, V. Frattolillo, S. Colosimo, A. Perrotta, A. Di Sessa, S. Guarino, E. Miraglia del Giudice, and P. Marzuillo, "Basal knowledge in the field of pediatric nephrology and its enhancement following specific training of chatgpt-4 "omni" and gemini 1.5 flash," *Pediatric Nephrology*, pp. 1–7, 2024.

[196] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023.

[197] G. Hinton, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[198] S. M. Magdy, F. Alwajih, S. Y. Kwon, R. Abdel-Salam, and M. Abdul-Mageed, "Gazelle: An instruction dataset for arabic writing assistance," *arXiv preprint arXiv:2410.18163*, 2024.

[199] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 595–46 623, 2023.

[200] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen,

C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[201] S. Rezayi, Z. Liu, Z. Wu, C. Dhakal, B. Ge, C. Zhen, T. Liu, and S. Li, "Agribert: Knowledge-infused agricultural language models for matching food and nutrition." in *IJCAI*, 2022, pp. 5150–5156.

[202] J. Kim, J. H. Lee, S. Kim, J. Park, K. M. Yoo, S. J. Kwon, and D. Lee, "Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Red Hook, NY, USA: Curran Associates, Inc., 2023, pp. 36 187–36 207.

[203] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Red Hook, NY, USA: Curran Associates, Inc., 2023, pp. 10 088–10 115.

[204] K. Inoue, K. Furukawa, I. Kobayashi, and H. Nabeshima, "Discovering rules by meta-level abduction," in *Inductive Logic Programming: 19th International Conference, ILP 2009, Leuven, Belgium, July 02-04, 2009. Revised Papers 19*. Springer, 2010, pp. 49–64.

[205] I. Kobayashi and K. Furukawa, "Hypothesis selection using domain theory in rule abductive support for skills," 2009, in: SIG-SKL (Skill Science). Japanese Society for Artificial Intelligence (in Japanese).

[206] Z. Li, Y. Cao, X. Xu, J. Jiang, X. Liu, Y. S. Teo, S.-W. Lin, and Y. Liu, "Llms for relational reasoning: How far are we?" in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 119–126.

[207] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati, "Large language models still can't plan (a benchmark for llms on planning and reasoning about

change)," in *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[208] A. Talmor, J. Herzig, N. Lourie, and J. Berant, "Commonsenseqa: A question answering challenge targeting commonsense knowledge," *arXiv preprint arXiv:1811.00937*, 2018.

[209] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[210] A. Ishay, Z. Yang, and J. Lee, "Leveraging large language models to generate answer set programs," *arXiv preprint arXiv:2307.07699*, 2023.

[211] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso *et al.*, "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models," *arXiv preprint arXiv:2206.04615*, 2022.

[212] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[213] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[214] S. J. Han, K. J. Ransom, A. Perfors, and C. Kemp, "Inductive reasoning in humans and large language models," *Cognitive Systems Research*, vol. 83, p. 101155, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389041723000839

[215] Z. Yang, L. Dong, X. Du, H. Cheng, E. Cambria, X. Liu, J. Gao, and F. Wei, "Language models as inductive reasoners," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, Y. Graham and M. Purver, Eds. St. Julian's, Malta: Association for Computational Linguistics, Mar. 2024, pp. 209–225. [Online]. Available: https://aclanthology.org/2024.eacl-long.13/

[216] R. Wang, E. Zelikman, G. Poesia, Y. Pu, N. Haber, and N. D. Goodman, "Hypothesis search: Inductive reasoning with language models," *arXiv preprint arXiv:2309.05660*, 2023.

[217] P. Tarau, "On llm-generated logic programs and their inference execution methods," *arXiv preprint arXiv:2502.09209*, 2025.

[218] J. P. Gandarela, D. S. Carvalho, and A. Freitas, "Inductive learning of logical theories with LLMs: A complexity-graded analysis," *arXiv preprint arXiv:2408.16779*, 2024.

[219] D. Cunnington, M. Law, J. Lobo, and A. Russo, "The role of foundation models in neuro-symbolic learning and reasoning," in *Neural-Symbolic Learning and Reasoning*, T. R. Besold, A. d'Avila Garcez, E. Jimenez-Ruiz, R. Confalonieri, P. Madhyastha, and B. Wagner, Eds. Cham: Springer Nature Switzerland, 2024, pp. 84–100.

[220] S. De Giorgis, A. Gangemi, and A. Russo, "Neurosymbolic graph enrichment for grounded world models," *arXiv preprint arXiv:2411.12671*, 2024.

[221] I. Kareem, K. Gallagher, M. Borroto, F. Ricca, and A. Russo, "Using learning from answer sets for robust question answering with llm," in *Logic Programming and Nonmonotonic Reasoning*, C. Dodaro, G. Gupta, and M. V. Martinez, Eds. Cham: Springer Nature Switzerland, 2025, pp. 112–125.

[222] M. Law, A. Russo, and K. Broda, "The ilasp system for inductive learning of answer set programs," *arXiv preprint arXiv:2005.00904*, 2020.

[223] A. Creswell, M. Shanahan, and I. Higgins, "Selection-inference: Exploiting large language models for interpretable logical reasoning," *arXiv preprint arXiv:2205.09712*, 2022.